The Analyzable Product Model Representation to Support Design-Analysis Integration

A Thesis
Presented to
The Academic Faculty

by

Diego Romano Tamburini

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Mechanical Engineering

Georgia Institute of Technology
May 1999

The Analyzable Product Model Representation to Support Design-Analysis Integration

Approved:

_____

Robert E. Fulton, Chairman

_____

Russell S. Peak, Co-Chairman

_____

Charles M. Eastman

_____

Anthony J. Gadient

_____

David W. Rosen

_____

Suresh K. Sitaraman

Date Approved: _____

DEDICATION

To the memory of my dear father, Libero Tamburini
(March 25th, 1925 – January 5th, 1999)

PREFACE

The current generation of CAD/CAM/CAE systems provides reliable, proven and sophisticated solutions for engineering mechanical products. However, we at the Engineering Information Systems Laboratory at the Georgia Institute of Technology believe that more work is needed on *integrating* these tools. More specifically, our research efforts focus on the integration issues that arise when information is shared between *design* and *analysis* applications.

As a member of this team, my job was to focus on the "product model side" of the design-analysis integration problem (see Figure 97-3) and propose a mechanism to model product information in a way that is more amenable for analysis. This thesis introduces a new representation of engineering products - termed Analyzable Product Model (APM) - aimed at facilitating design-analysis integration. APMs provide a stepping stone between design and analysis representations which absorbs much of the complexity that would be otherwise passed to analysis applications, resulting in leaner and easier to maintain analysis applications. This APM Representation complements the Multi-Representation Approach (MRA) developed at the Georgia Institute of Technology by Drs. Russell S. Peak and Robert E. Fulton (Peak 1993; Peak and Fulton 1993c; Peak, Fulton et al. 1998) (see Chapter 7) by providing the product information required by their Product Model-Based Analysis Models (PBAMs), thus filling the gap between design tools and PBAMs. Together, the APM Representation and the MRA provide a highly modular and flexible design-analysis architecture.

This research benefited enormously from a significant amount of exposure to real-world applications during the projects in which I had the privilege to participate. In many occasions, the issues that arose in these projects steered the direction of my research. These projects also provided me with invaluable test cases with which I was able to test the validity and applicability of the concepts I was developing.

During my research I produced a significant amount of prototyping code. In writing this thesis, I tried to achieve a balance between presenting the concepts in a formal and generic manner and providing implementation examples and pieces of actual code. Although I made it a point not to stress coding too much, in several occasions I found it useful to show some code to illustrate the concepts and provide a better idea of how they could be implemented. I did not include, however, every single line of code I wrote for this research. For the interested reader, in Chapter 64 I provide references to web pages in which the complete code of the prototypes can be found.

This work, of course, does not provide a definitive solution to the problem of design-analysis engineering. I hope, however, that it does provide an initial step in the right direction. It is very rewarding to see that members of the EIS Lab team are currently working on refining and enhancing the APM Representation (as summarized in Subsection 113), and exposing it to further testing in the projects they participate.

Organization of this thesis

This thesis is organized into nine chapters: Chapter 1 provides a general introduction to the problem of design-analysis integration, heterogeneous transformations and idealizations. Chapter 7 surveys some related research efforts in this area and identifies several gaps in the current state of this field that require further attention. These two chapters establish the groundwork for Chapter 27, which formally states the problem addressed by this research and lists the objectives that drove the development of the Analyzable Product Model Representation. Chapter 38 formally introduces the APM Representation. This is the core chapter of this thesis, in which the theoretical contribution of this work is presented. Chapter 64 describes a prototype implementation of the APM Representation developed by the author for this work. Chapter 83 presents a series of test cases that utilize the prototype implementation in real-world applications. Chapter 97 evaluates the results of Chapter 83 to assert to what extent the APM Representation met the research objectives stated in Chapter 27. Chapter 110 recommends several extensions to this work aimed at overcoming the limitations and unfulfilled objectives identified throughout the thesis. Chapter 114 wraps up this work by providing some concluding remarks and thoughts.

ACKNOWLEDGEMENTS

During the long journey towards my Ph.D., I was blessed with the invaluable help and encouragement from many friends, colleagues, and members of my family, to whom I will attempt to express my deepest gratitude in the paragraphs that follow.

First, I must start thanking my dear advisor, Dr. Robert E. Fulton for sharing his technical knowledge and vast experience with me. I also want to thank him for his constant moral support and encouragement, and for giving me the opportunity to participate in real, meaningful projects that enriched my experience as a Ph.D. student. I hope that during my time as his student I was able to assimilate at least a small portion of his wisdom and passion for research and engineering.

I also want to thank Dr. Russell S. Peak for his continuous guidance, support and tutoring throughout my research and during the production of this thesis. I also thank him for his thoroughness and insightful comments and for helping me keep the quality of this thesis always in check.

I would also like to thank the other distinguished members of my thesis committee: Drs. Charles Eastman, Anthony Gadient, David Rosen and Suresh Sitaraman for their time, patience and valuable feedback.

Being part of the Engineering Information Systems Laboratory at Georgia Tech was both an honor and a real pleasure. I would like to thank my fellow members of the team: Angela Birkes, Ashok Chandrasekhar, Selçuk Cimtalay, Tal Cohen, Neil Hall, Chien Hsiung, Haruko Peak, M.C. Ramesh, Andy Scholand, Miyako Wilson and Wen Zhou for their constant support and friendship. I learned a lot from them, and will always regard them as my research brothers and sisters for the rest of my career. I would also like to thank Donna Rodgers for her continuous help and for being so supportive with the numerous administrative issues that arose during my studies.

I was very privileged for having the opportunity to participate in several industry-sponsored projects that enriched my Ph.D. experience – and this thesis - enormously. I would like to extend my gratitude to those companies and individuals that provided their support and guidance for these projects. From the TIGER project: Lynwood Hines, Gerry Graves and Mike Stiteler (SCRA); Ken Buchannan and Darla Nettles (Arthur D. Little); Adam Owsley (ITI); Brian Anderson and Jeff Lewis (Holaday Circuits); David Christenson, Mike Keenan and Greg Smith (Boeing). From the Subsystem Interface Integration (SII) Project: Marcia Herndon and Steve Howard (Lockheed Martin Aeronautical Systems). From the Product Simulation Integration (PSI) Structures Project: Hugo Korwaser and David Leal (CAESAR Systems); Mats Kjellberg, Mats Nilsson, Nigel Shaw and Staffan Westbeck (EuroSTEP); Jack Blaylock, Kal Brauner, Dave Briggs, Richard Brooks, Rod Dreisbach, Dave Jack, Lee Krueger, Dan Kwong, Bob Masters, Dale Novstrup, Marty Prather, Frank Robl, Joe Sharp, Bob Thomas, Scott Tsao, Ching-Yang Wang, Peter Wilson and King Yee (Boeing Commercial Airplane Group).

Special thanks to my colleagues at SDRC (Craig Baldridge, John Bretl, Daniel Caeton, Larry Epp, Frances Evans, Gintas Jazbutis, Yogish Kode, Chandru Narayan, Booyong Oh, Ravi Rangan, Laura Readdy, Amy Strucko, Ramesh Venugopal, Bruce Winegarden, Gerard Woods, Michael Zawacki, David Zhou) for giving me the opportunity to be a member of their team despite the fact that I had not yet concluded my doctorate. Even though juggling between my thesis and my job was a tough task, it turned out to be a great incentive to finish. I thank them for their vote of confidence and patience.

Words are inadequate to express how thankful I am to my family. Babbo and Mamma, your love, encouragement and example gave me the strength to endure this difficult effort. This thesis - and the accomplishment it represents - are truly yours. Babbo, even though you are not physically among us anymore, I hope you can too celebrate this achievement and be proud of your son. I will miss you dearly. Donatella, Janet, Rebeca and Ricardo, I thank you for your constant support and encouragement and for being such a great and supportive family to me. Special thanks also to my dear friends Romer Bracho, Paulo Flor, Pier Luigi Michelangelo, Gregory Mitchell, Gerardo Reyes, Mario Salvatore and Daniel Stamft for their continuous support and for always making me feel at home, and to Carlos Juan Reyes, whose encouragement was pivotal in my decision to come to the United States to pursue my graduate studies, and who has always looked after me since then.

I want to especially thank Patricia, my wife, for her love, patience, and sacrifices during these very long six years. Thank you for putting your life on hold for me and for your determined support. Without all those words of encouragement, back rubs, meals, and cups of coffee you brought me while I was stuck in front of the computer writing this thesis I would have probably given up long ago. Finally, I would like to thank (as unusual as it may seem!) my dear dog Pistacho. Thank you for your simple, pure, and unconditional love that helped me keep things in perspective and for those early mornings in which you walked into my office - almost asleep - to "check up on me" while I was working.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Despite the number of sophisticated CAD/CAE tools available, collecting the product information needed for engineering analysis often poses a significant challenge. Contributing to this is the fact that there is rarely an integrated source of analysis information, since the product development normally involves designers from several disciplines using a variety of independent computing and manual systems. In addition, analysis models need idealized product information, which may require significant simplification or transformation of the design data. Some point-to-point solutions exist that integrate specific design and analysis tools, but the knowledge used to combine and idealize design information for analysis purposes is normally not captured in an explicit reusable and traceable form.

This thesis introduces a new representation of engineering products - termed Analyzable Product Model (APM) - aimed at facilitating design-analysis integration. This representation defines formal, generic, computer-interpretable constructs to create and manipulate analysis-oriented views of engineering products. These views help bridge the semantic gap between design and analysis representations, providing a unified perspective more suitable for analysis which multiple analysis applications can share. They are obtained by merging design representations from multiple sources and adding idealized information.

This thesis presents test cases and a prototype implementation used to validate the APM Representation. These test cases, which come from the electronic packaging and aerospace industries, utilize commercial CAD/CAE tools and STEP information exchange standards.

As these test cases demonstrate, APMs provide a stepping stone between design and analysis which absorbs much of the complexity that would be otherwise passed to analysis applications, resulting in leaner analysis applications. Another key APM distinctive demonstrated is the ability to formally represent the knowledge required to combine and idealize design information for analysis. While such knowledge is critical to achieving repeatable and automatable analysis, it is largely lost today.

CHAPTER 1

DESIGN-ANALYSIS INTEGRATION BACKGROUND

Design-Analysis Integration Problem Overview

During the computer-aided development of a product, the primary task of design engineers is to create a detailed description of the product that contains enough information to support the requirements of the different stages of its life cycle. At certain intermediate points of the product development cycle, this design representation is used to drive a series of engineering analyses that validate the design against several criteria and help predict the physical behavior of the product under various conditions. In order to perform these analyses, the design representation must be first idealized and transformed into some form that admits mathematical evaluation. This form is normally referred to as "analysis representations", "analysis models" or, more specifically, "product model-based analysis models" (Peak, Fulton et al. 1998) (to differentiate them from *generic* analysis models, which are not linked to any particular product). Computer programs called "analysis applications" provide the necessary interfaces to enable user interaction with electronic forms of these analysis models. The results of these engineering analyses are used to successively refine the design representation. Figure 1-1 illustrates this design-idealize-analyze sequence using a simple mechanical component (a linkage) as example.

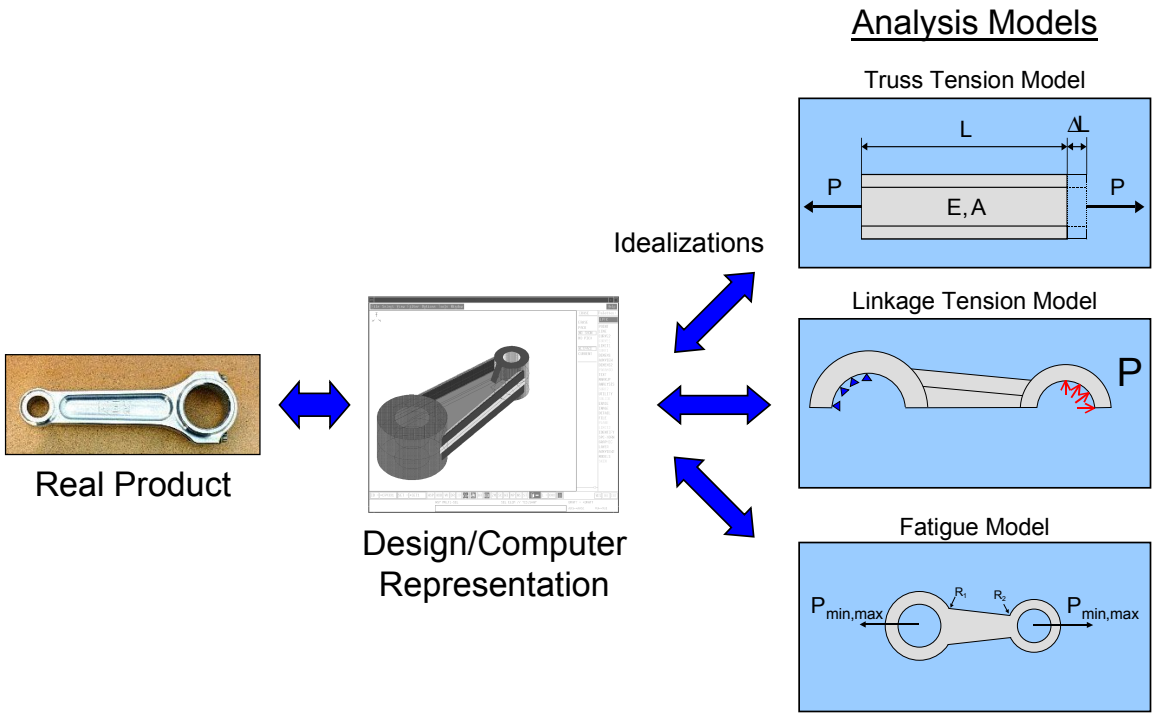


Figure 1-1: Design-Analysis Integration Example

Although both the design and the analysis representations described above are views of the same product, they describe it at very different levels of semantic content, and obtaining the latter from the first is generally a difficult task. Hence, design-analysis integration often turns out to be a difficult proposition. The current generation of CAD/CAM/CAE systems provides very strong solutions for engineering mechanical products, but each does so with proprietary technical capabilities, and worse, often with proprietary data formats not accessible by other applications (Al-Timimi and MacKrell 1996). The unstructured development of these systems over the years has made it difficult to integrate both the systems themselves and the information they manipulate (Brooke, Pennington et al. 1995). As a result, even though there is a large number of sophisticated computer aided engineering tools available, the current status is that in general design and analysis software tools are not compatible enough to exchange data directly - without cumbersome (manual or semi-automatic) transformation (Kemper and Moerkotte 1994). In many cases, data needed by the analysis models has to be manually retrieved and re-inputted in some other computer application for analysis. In addition, due to the large syntactic and semantic gap between design and analysis representations, some raw design information must undergo significant

transformation, simplification and/or idealization before being fed into the analysis models on which the analysis applications are based (Armstrong 1994; Shephard, Korngold et al. 1990). This is usually a tedious, slow, and error-prone process that characterizes the infamous "islands of automation".

Added to these incompatibility problems is the fact that, in a real scenario, the development of a product requires participation of designers from several disciplines who use a wide variety of independent software systems. These multiple design applications generate a large and complex aggregation of diverse design information, scattered across several data sets with different, often proprietary, formats and data structures. As a result, there is rarely a single, integrated source of analysis information readily available. Integrating the information contained in these disjoint sources of design data requires a significant amount of engineering knowledge. Moreover, this information is often both redundant and incomplete for analysis purposes. Figure 1-2 illustrates this multiple design and analysis applications scenario.

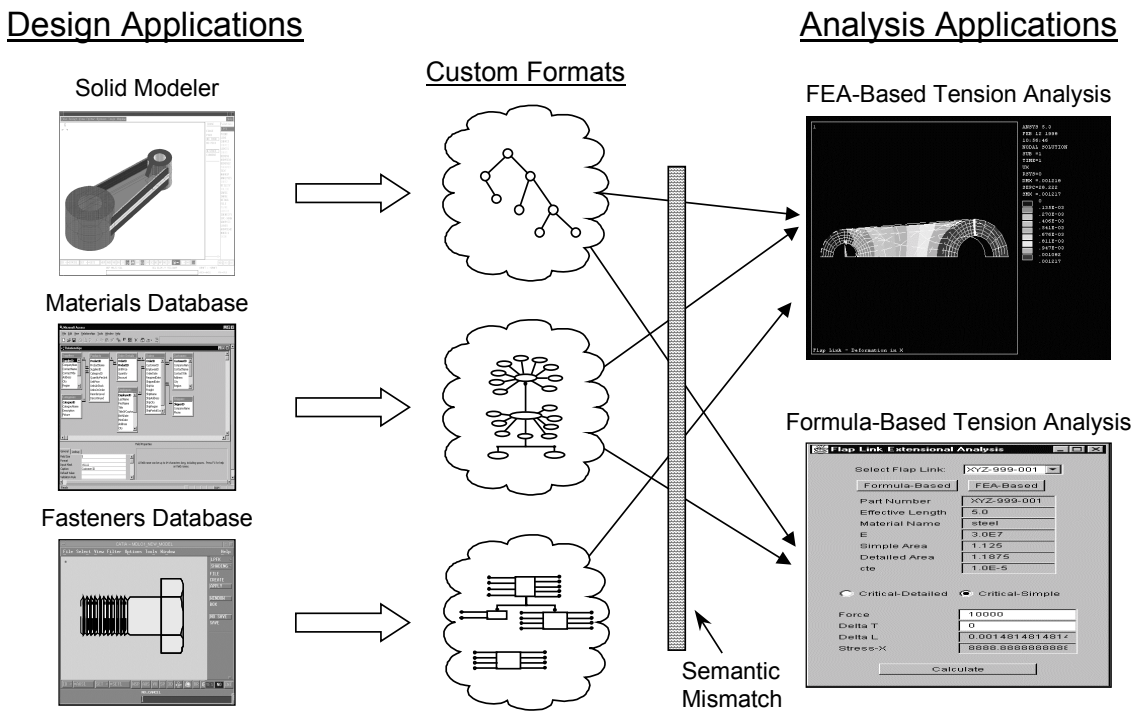


Figure 1-2: Multiple Design and Analysis Applications Scenario

Design-analysis integration is enjoying increasing attention because engineering design firms are adopting the strategy of giving designers the tools needed to predict the performance of a design, rather than just to define its dimensions (Deitz 1997). In other words, they are now turning designers into analysts (to some degree). This of course imposes new requirements for the developers of CAD/CAE systems, as they now have to provide tighter integration between design and analysis functions. The ultimate goal is to enable designers to perform analysis directly from their CAD tools (by making analysis functions available via the CAD system's user interface) thus reducing the time required to prepare a model for analysis.

Idealization and Synthesis

In engineering terms, to *idealize* is to construct an abstracted model of the real system that will admit some form of mathematical analysis (Shigley and Mischke 1989). Most frequently, idealization refers specifically to the transformations that are applied to the *design* representation of a part, which is already an idealized version of the "real" or "physical" part in that the design representation is a model of the typical actual part (as illustrated in Figure 1-1). Idealizations are applied to design information because most problems contain complexities that render numerical simulation difficult or impossible to analyze. In addition, it is usually neither feasible nor desirable to analyze in detail all aspects of a product because of its inherent complexities. Thus, in practice, certain complexities can be simplified in order to make numerical computation more efficient (or possible) and some redundancies can be ignored without drastically affecting the accuracy of the analysis. Idealization techniques can be applied to any of the following aspects of a physical system: geometry domain, phenomena, boundary conditions, initial conditions, material properties or mathematical equations (Finn, Grimson et al. 1992). As Finn points out, the major challenge to the engineer is identifying the importance of different systems aspects, performing the appropriate simplifications or idealizations and finally assessing the suitability of the resulting model for analysis. Finn provides the following categorization of engineering idealizations[1]:

[1] Finn distinguishes between *simplifications* and *idealizations*. In the list below, he considers the first six operations simplifications and the last three idealizations. For the purposes of this discussion, a simplification will be considered as a type of idealization.

- ***Dimensional Reduction:*** involves reducing the degree of spatial analysis or time analysis. Spatial analysis may involve reduction from 3-dimensional to 2-dimensional or 1-dimensional analysis. Time analysis may involve reducing a transient analysis to a quasi-static or steady state analysis.
- ***Geometric Symmetries:*** involve removing redundant domains by identifying spatial symmetries and applying compensatory boundary conditions.
- ***Feature Removal:*** involves removing some engineering feature that is not expected to contribute significantly to the overall analysis results (for example a small hole or a fin).
- ***Domain Alteration:*** involves changing some aspect of the spatial domain so that the analysis is simplified (for example, modeling a thin aerofoil as a thin plate).
- ***Phenomenon Removal:*** involves the removal from analysis of complete phenomena based on the decision to ignore the effect of that phenomenon (for example, ignoring stress effects within the physical system).
- ***Phenomenon Reduction:*** applies to situations where a multi-component phenomenon exists and a particular component is removed is removed because its significance is judged to be of minor importance (for example, removing radiation analysis from a heat transfer problem).
- ***Phenomenon Idealizations:*** involve the use of mathematical expressions to describe the system phenomena. For example, in fluid analysis, a number of mathematical equation models are available to solve for flow analysis: parallel flow can be modeled using the full Navier Stokes equations or a Couette flow model.
- ***Boundary Condition Idealizations:*** may involve applying a mathematical equation to model a boundary condition that does not perfectly represent the physical boundary conditions. For example, in heat transfer modeling, a non-ideal surface may be modeled as a black body or gray body surface.

- ***Material Idealizations:*** generally involve the use of idealized material laws to model some complex material behavior. For example, modeling an expected non-linear material response using a linear approximation function.

Synthesis is the opposite of idealization; the act of "appearing as a material form or taking substantial shape", that is, going from an abstract or ideal representation to a physical representation. Effectively, synthesis is performed in three steps: the first is to decide on the variables (primitive or complex) from the design representation of the part that are going to be populated with values. The second step is to assign values to these variables. The third step is to use this populated design representation to actually create or manufacture the physical part[2]. The assignment of values to the design variables is normally based on the results of engineering analyses, but it could possibly be based on rules-of-thumb, experience or even arbitrary judgement. Synthesis is a more complex process than idealization because the design representation of the product is richer than the abstract representation, and therefore it may be necessary to add information (such as additional constraints or pre-determined design configurations) in order to go from the abstract to the physical. Additionally, product-analysis transformations that have a closed-form solution in one direction (for instance, a relation of the form $A_1 = \delta(P_1, P_2)$, where A_1 is an idealized variable, and P_1 and P_2 are two product variables), may not have one if, for example, we need to solve for one of the product variables, say P_1.

The design representation of a product is expressed exclusively in terms of *product variables*, whereas analysis representations are expressed as a combination of product variables and *idealized variables*. Product and idealized variables are related by *product idealization relations* (Peak, Fulton et al. 1998). When these product idealization relations are used to obtain idealized variables from product variables (that is, in their "forward" form) they are called idealizations. When they are used in the "reverse" direction, that is, to obtain product variables from idealized variables, they are called synthesis relations. In the context of design-analysis integration, idealization and synthesis characterize the bi-directional nature of the design-analysis process: idealization is used when the design description of the product is abstracted to prepare it for analysis, whereas synthesis is used when the results of the analyses are used to make changes in the design (as in optimization).

[2] In this thesis, the focus will be on the first two steps.

Homogeneous and Heterogeneous Data Exchange

Most of the discussions about data exchange between engineering systems focus on *homogeneous* data exchange cases. In general, homogeneous data exchanges occur between systems that are similar in scope and semantics, hence mostly requiring syntactic translation of the data. Homogeneous data exchanges normally take place between systems that have (Al-Timimi and MacKrell 1996):

- The same data model: for example, between two solid modelers from the same vendor or from two solid modelers from different vendors exchanging data through some standard data exchange model such as STEP AP203 (Appendix A).
- Different data models but same level of richness, scope and semantics: there is normally a direct mapping from one model to the other that can potentially be automated. For example, when two solid modelers from different vendors exchange information about 2D circles; one solid modeler may represent 2D circles using three points and the other using a point and a radius. This type of data exchange is mostly a syntactic translation process that requires a customized translator between each pair of systems.
- Data models with different levels of richness but same scope and semantics: naturally, in this case, the data exchange goes most easily from the system with the higher level of richness to the system with the lower level of richness. For example, from a solid modeler to a surface or wireframe modeler. This is also mostly a syntactic translation process that can be automated.

However, one of the main differentiating characteristics of the exchange of information between design and analysis is its *heterogeneous* nature. This heterogeneity is caused by the large gap in scope and semantics that exists between design and analysis representations (Peak 1993), which requires a syntactic *and* a semantic transformation of the data being exchanged. For example, as illustrated in Figure 1-3, an Electrical CAD (E/CAD) system and a finite element analysis system may describe the same Printed Wiring Assembly (PWA) from very different points of view: the first describes it in PWA-domain terms (components, traces, layers, pads, etc.) and the second in terms of nodes and elements.

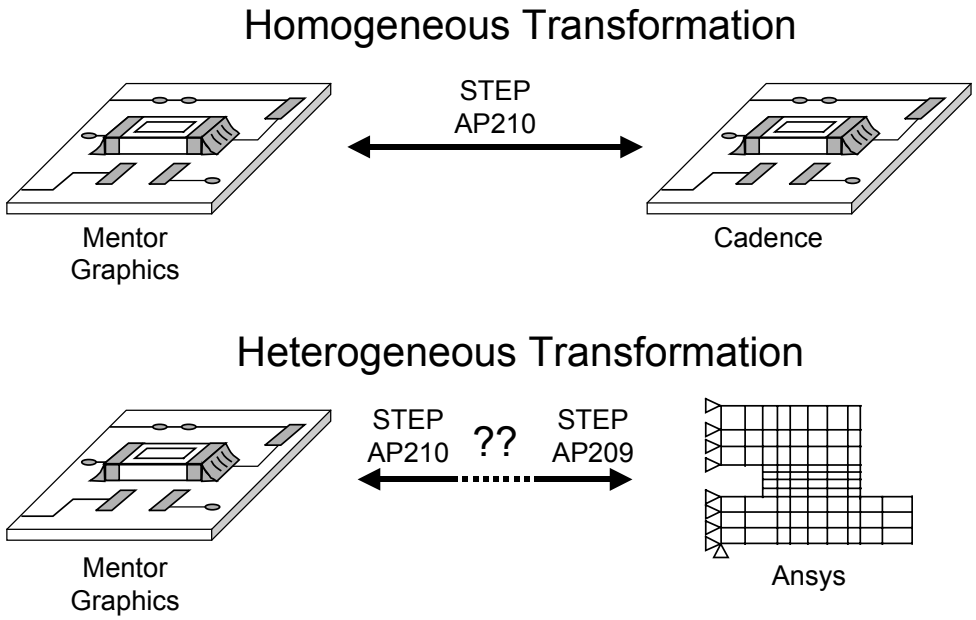


Figure 1-3: Homogeneous and Heterogeneous Data Exchange

Although the level of detail required by analysis models is usually lower than the one required by design models, an analysis model is more than just a simplified design model. In addition to simplified design information, analysis models require idealized information about the product that is not normally contained in design representations. Therefore, rules defining how this additional idealized information is merged with design information have to be supplied as part of the mapping specification.

In addition, this transformation process may be dependent on the *values* of the attributes in the design model. To illustrate this, consider the example of a geometric model of a plate with a hole that is being fed into a stress analysis application. The mapping between the geometric representation of the plate and its analysis representation may specify that the hole may be ignored if the ratio between its diameter and the length of the plate is smaller than certain value. This conditional information also needs to be captured explicitly in order to enable the exchange of data.

Finally, another characteristic of heterogeneous transformations between design and analysis representations that makes design-analysis integration especially challenging is that these transformations can take place at different levels of fidelity. For example, the heterogeneous

transformation between a CAD model and a FEA model can occur at two levels of fidelity: one if the FEA model is 2-D and another if it is 3-D. The term *Multi-Fidelity Heterogeneous Transformations* can be used to convey this notion.

Information Requirements of Design-Analysis Integration

Design-analysis integration has some unique characteristics that impose some special requirements on the data exchange. Among these requirements are:

- ***Multiple sources of data:*** data needed for analysis usually spans multiple design repositories and is generally stored in a variety of formats. This is especially true for multidisciplinary products like PWAs that involve E/CAD as well as M/CAD tools. For example, the analysis of PWB bending requires data about the layout of the PWB (created with an E/CAD tool), data about its detailed geometry (created with a M/CAD tool), and data about the manufacturing process (process temperatures, forces, etc., created by a process/factory definition tool). Another example is the finite-element analysis of a mechanical component, which requires information about the geometry of the component – normally created with a solid modeler – and about the properties of the material of which it is made – created and maintained in a materials data management system. Integrating the information from multiple sources requires a significant amount of engineering knowledge, which needs to be captured explicitly if the data exchange between design and analysis is to be automated.
- ***Reusable product idealization relations:*** as discussed above, product idealization relations relate detailed, design-oriented attributes with idealized, analysis-oriented attributes (Peak, Fulton et al. 1998). As shown in Figure 1-4, multiple product idealizations may be applied to a given product, and a given product idealization may be used potentially by more than one analysis application. These product idealization relations also need to be captured explicitly in order to automate the data exchange between design and analysis. These relations may be relatively complex, involving non-linear expressions, transcendental functions, conditional statements, iterations, etc.

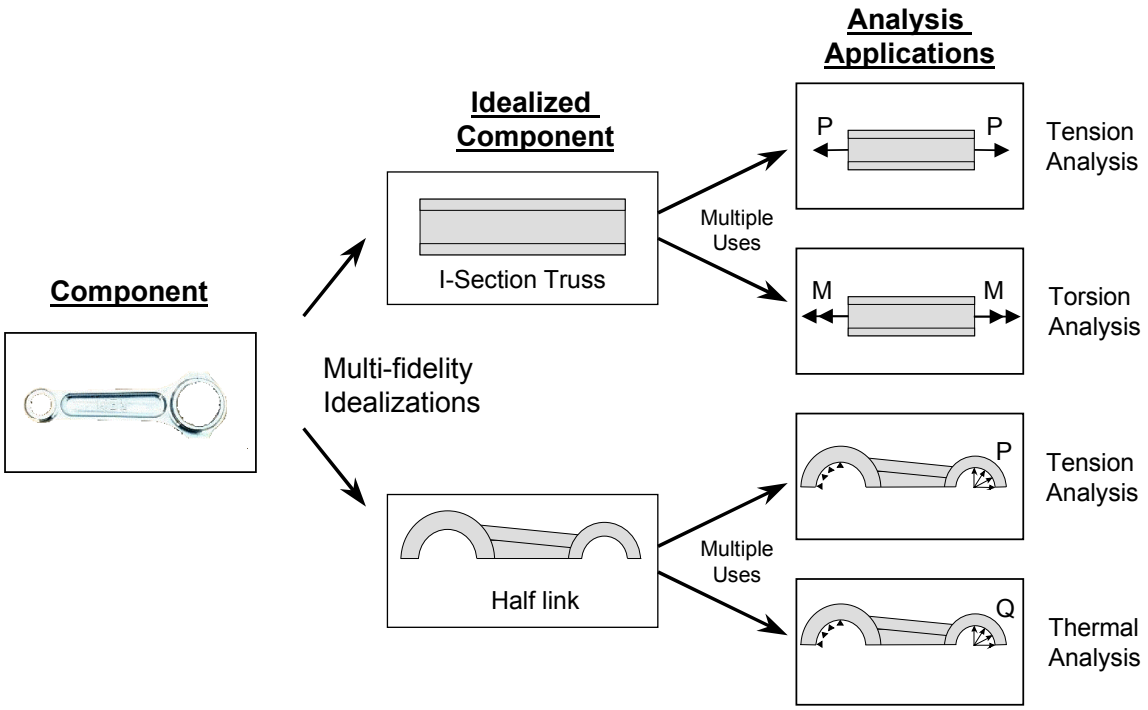


Figure 1-4**:** Reusable Multi-Fidelity Product Idealizations

- ***Multi-directional product idealization relations:*** the input and outputs in a given product idealization relation may vary according to the particular design-analysis scenario. Figure 1-5 shows three common situations involving product idealization relation **I** and analysis model **A** (used to calculate the axial stress Γ of the plate when an axial load **P** is applied). In this figure, attributes **width**, **thickness** and $\mathbf{d_1}$ (the width, thickness, and diameter of hole 1 of the plate shown on the left side of the figure) are design attributes and attribute $\mathbf{A_C}$ (the critical area of the plate) is an idealized attribute. In the first situation – termed *design checking* – values for **width** (20 in), **thickness** (0.25 in) and $\mathbf{d_1}$ (7.5 in) are entered as inputs to relation **I** to obtain the value of $\mathbf{A_C}$ (3.125 in^2). The obtained value of $\mathbf{A_C}$ and a value of **P** (100 lb) are entered to the analysis model **A** to obtain the axial stress Γ (32 psi). In the second situation – termed *iterative synthesis* – desired analysis results are entered first (Γ = 30 psi when **P** = 100 lb) in order to obtain a target value of $\mathbf{A_C}$ (3.33 in^2). Then relation **I** is used to iterate over the value of $\mathbf{d_1}$ until the target value for $\mathbf{A_C}$ is reached, resulting in a value $\mathbf{d_1}$ =

6.66 in. In the third situation – termed *synthesis* – the desired analysis results are also entered first, but this time relation **I** is used "in reverse" to obtain the value of $\mathbf{d_1}$ (without having to iterate). Note that in this last case, relation **I** is used in a different direction than in the first case (that is, the input/output combination is different).

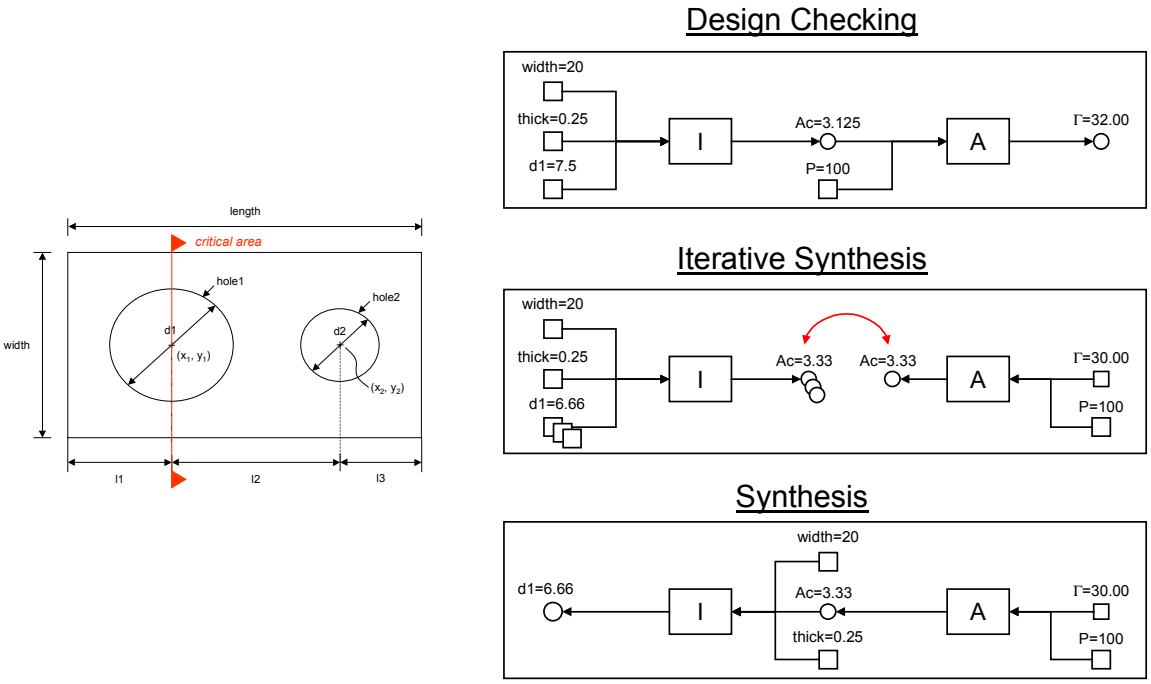


Figure 1-5: Multi-directional Product Idealization Relations

- ***Unavailable Analysis Data:*** design data often must be complemented with additional data to perform analysis. Some analyses may need very specific information that is not being supplied by any of the design tools or that is not readily available in any form. For example, a PWB warpage analysis may need detailed information about the layup of the board. However, this information is not provided by the electrical engineer that designs the PWA because he or she is not concerned with that level of detail.
- ***Simplifications:*** much of the data representing the design of the complete product is not used at all in the analysis (Morris, Mitchell et al. 1992). A common example of this

is geometry; analysis models rarely need all the detailed information used by geometric modelers to represent the geometry of the product and therefore normally use a simplified version of the real geometry.

- ***Data complexity:*** engineering analyses tend to be "information-hungry"; they normally demand a large number of *types* of data - complicatedly interconnected - as opposed to a large number of instances of each type of data.
- ***Variety of types and multi-fidelity of analyses:*** in most cases (Figure 1-4), each phenomenon requires a separate analysis application to predict its effects (Brooke, Pennington et al. 1995). However, the problem is not only the *number* of analyses that need to be supported but also the *variety* of analyses and their information requirements (for example, structural versus thermal analyses). Moreover, a given analysis may be performed using several solution methods (e.g., formula-based, finite-element analysis, etc.) and/or at multiple levels of fidelity for the same phenomenon, the information requirements varying from one solution to another. The choice of a particular combination of analysis model and solution method will depend on the level of accuracy desired and the computer resources available, keeping in mind that, in general, there is a tradeoff between the level of fidelity used and the computation cost. For example, a rough analysis model may be sufficient during early design, leaving the usage of an analysis model with a higher level of fidelity for when more accurate results are needed.
- ***Multiple Levels of Product Structure:*** the same design model may be viewed, for analysis purposes, at different levels of detail. For example, Figure 1-6 illustrates two views of the same assembly: one focusing on a particular feature and the other utilizing the entire assembly.

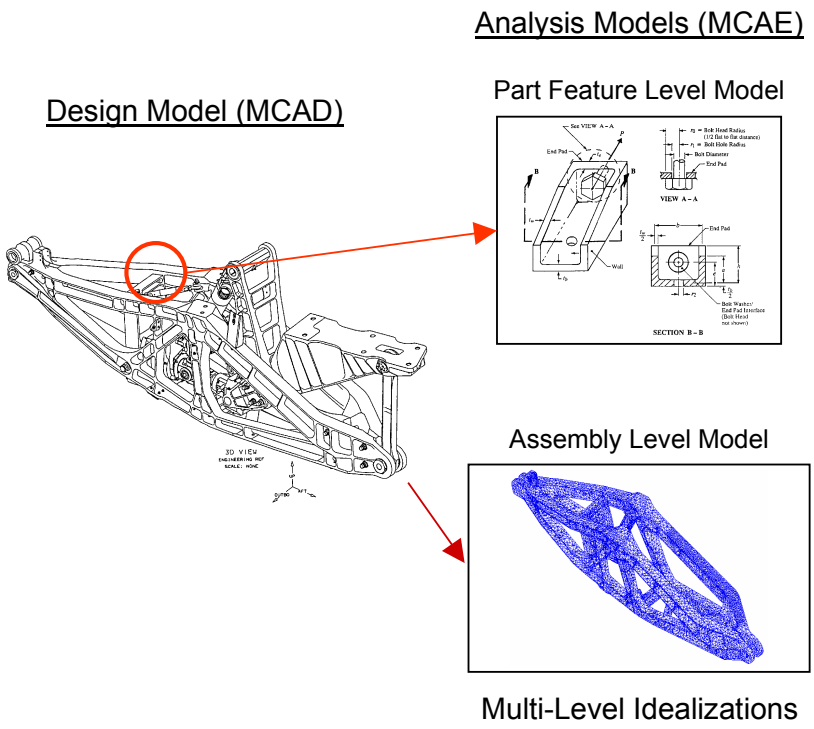


Figure 1-6: Multiple Levels of Product Structure

Design-Analysis Integration Using Neutral Product Data Exchange Standards

As discussed in Section 2, multiple design applications generate data about the product and store it in different, often proprietary, formats and data structures. A step forward towards facilitating design-analysis integration is the utilization of neutral exchange formats. In this approach, as shown in Figure 1-7, the data of each design application is translated to a neutral (standard) format. Analysis applications read the data from these standard formats without regard of the application that generated this data, eliminating the need for point-to-point translators and updating the analysis applications each time a new release of a design system is released.

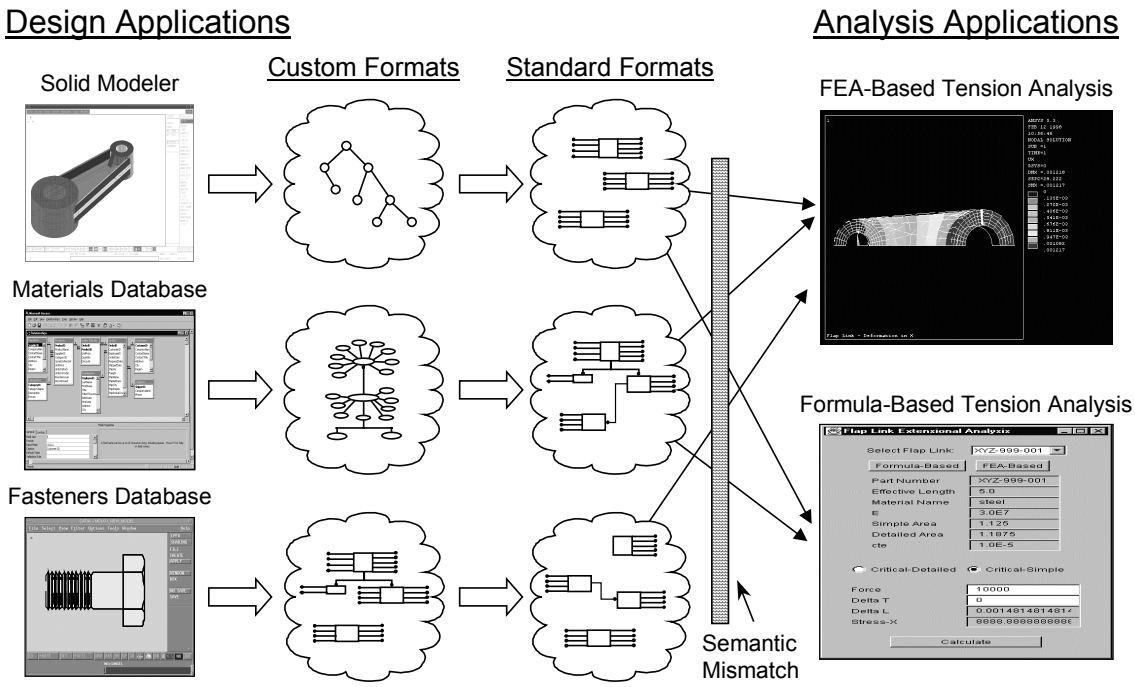


Figure 1-7**:** Design-Analysis Integration Using Neutral Data Exchange Standards

Perhaps the largest and most important present effort to develop neutral representations of product data is the Standard for the Exchange of Product Model Data (STEP – officially known as ISO 10303). Appendix A briefly overviews the STEP standard and explains the basic concepts that will be used in this thesis. For more comprehensive discussions on STEP the reader is referred to (Al-Timimi and MacKrell 1996; Hardwick 1994; ISO 10303-1 1994; Laurance 1994; Owen 1993). However, this approach alone is not sufficient due to the heterogeneous nature of CAD-CAE transformations (Figure 1-3).

CHAPTER 2

RELATED WORK

This chapter overviews several research activities in the area of design-analysis integration. The common objective of the works included in this survey is to provide some mechanism for translating or transforming product data from one representation to another, in order to support the needs of multiple, integrated computer-aided applications. Of particular significance to this thesis are those works that focus on transforming and idealizing product data to support engineering analysis, thus enabling design-analysis integration. Some of the projects overviewed here, however, do not specifically address the exchange of information between design and analysis representations but are included anyway because they provide valuable insight as to the available mechanisms to exchange data between representations in general.

This survey is grouped into three subsections: Subsection 8 overviews the design-analysis integration research activity at the Engineering Information Systems Laboratory (EIS Lab) in the George W. Woodruff School of Mechanical Engineering of the Georgia Institute of Technology, which includes some of the preliminary research that lead to the development of the concepts presented in this thesis; Subsection 14 overviews several related activities performed by other research groups; and Subsection 22 presents several design-analysis integration works involving STEP and overviews the engineering analysis standardization activities currently being performed by the international standardization community.

Design-Analysis Integration Research at the Engineering Information Systems Laboratory

The Georgia Tech EIS Lab (of which the author is a member) has been conducting research and participating in several industrial projects on design-analysis integration for several years. This section overviews some of these research efforts which are closely related to this thesis.

The Multi-Representation Architecture (MRA)

Peak (1993; 1993a; 1993b; 1993c; 1998) developed the *multi-representation architecture* (MRA, Figure 7-1), a design-analysis integration strategy that views CAD/CAE integration as an information-intensive mapping between design models and analysis models. Peak argues that the gap between design and analysis models is too large for a single general integration bridge, and therefore divides the MRA into four information representations that act as stepping stones between the design and analysis tool extremes. These four information representations are: solution method models (SMMs), analysis building blocks (ABBs), product models (PMs), and product model-based analysis models (PBAMs).

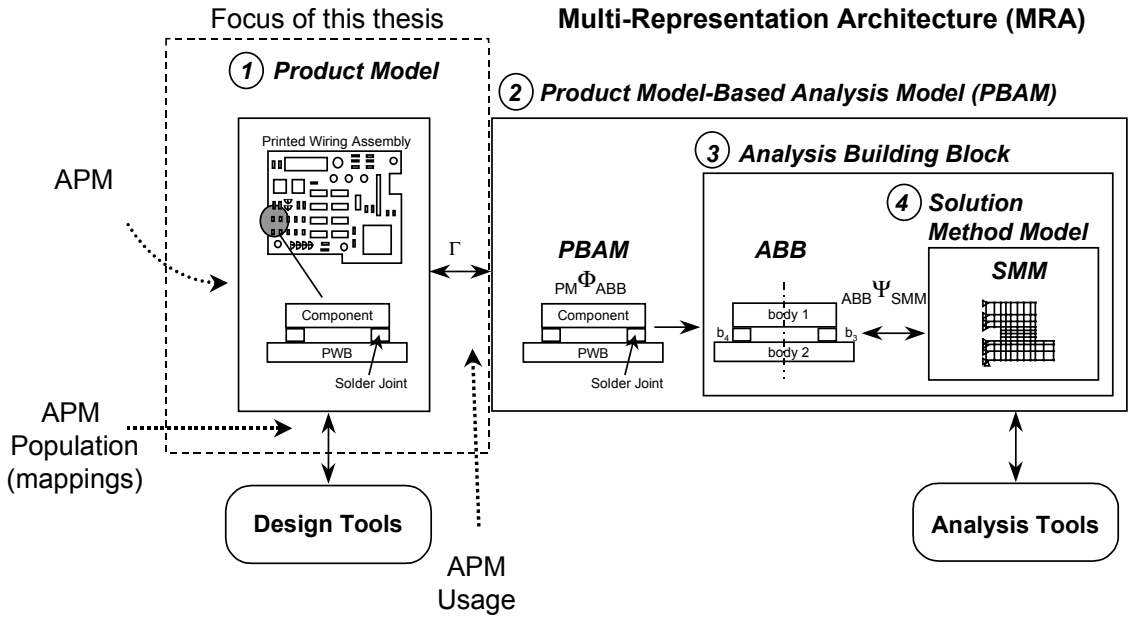


Figure 7-1: The Multi-Representation Architecture for Design-Analysis Integration

On the right extreme of the MRA (Figure 7-1) are *solution method models* (SMMs) representing analysis models in relatively low-level, solution-specific form. SMMs combine solution tool inputs, outputs and control into a single information entity (an object) to facilitate automated solution tools access and results retrieval. SMMs are object-oriented wrappers around solution tools (e.g., FEA systems) that utilize an agent-based framework to obtain analysis results in a highly automated manner. *Analysis building blocks* (ABBs)

represent engineering analysis concepts in a manner that is largely independent of product application and solution method. ABBs obtain results by generating SMMs through transformations (labeled ${}_{ABB}\Psi_{SMM}$) that are based on solution method considerations. *Product Models* (PMs, on the left extreme) represent detailed, design-oriented product information. A PM is considered the master description of a product, which supplies information to other life cycle tasks, including engineering analysis and manufacturing. To enable usage by potentially many analysis applications, PMs in the MRA go beyond their traditional role and support idealizations that relate detailed, design-oriented attributes with simplified, analysis-oriented attributes. Finally, *product model-based analysis models* (PBAMs) contain linkages (labeled ${}_{PM}\chi_{ABB}$) that represent design-analysis associativity between PMs and ABBs. These associativity linkages indicate the usage of idealizations for a particular analysis application. PBAMs have been used to create catalogs of ready-to-use analysis modules for applications such as solder joint deformation and fatigue, PWB warpage, and plated-through holes (Peak 1993; Peak and Fulton 1993b; Peak, Scholand et al. 1996).

From the MRA viewpoint, providing solutions to the design-analysis integration problem involves defining these four representations (SMMs, ABBs, PMs and PBAMs) and two inter-representation mappings (${}_{ABB}\Psi_{SMM}$ and ${}_{PM}\Phi_{ABB}$). The MRA achieves flexibility by supporting different solution tools and design tools, and by accommodating analysis models of diverse discipline, complexity and solution method.

Cimtalay (1996) introduces an optimization technique closely integrated with the MRA. In this technique, modular software entities called *optimization agents* use the analysis results obtained by PBAMs for design optimization, by plugging them into the objective and/or constraint functions of its internal optimization model and obtaining new design variables needed to reduce the objective function. These new design variables are fed back into the product model and the process is repeated until the objective function value converges. This technique enables a closed-loop process that improves designs by meeting some selected criteria and constraints. The designer can choose the proper optimization agent based on the complexity of the analysis, types of models and tool availability. The paper provides more details on how optimization agents are integrated with PBAMs and product models.

The Analysis-Oriented Product Model (AOPM)

In his MRA work, Peak focused on developing a mechanism for extracting and transforming data from an integrated product database in order to perform some engineering analyses. In other words, he focused mainly on the PBAM/ABB/SMM components of the MRA. He describes how PBAMs use product information and idealizations supported by product models and points outs that, to enable usage by potentially many analysis applications, product models in the MRA must go beyond their traditional role and support idealizations. However, he does not go into further detail on how these product models are created and populated with data generated by different design tools. In his prototypes, this product database was created and populated manually, assuming emerging standards like STEP would enable automated production of similar databases.

This thesis complements the MRA work by focusing on the Product Model component of the architecture. Hence, the MRA provided a contextual framework for the development of the concepts presented in this thesis. The first paper in the evolution of these concepts was (Tamburini, Peak et al. 1996), in which the author introduces for the first time the idea of an integrated, object-oriented representation that is populated with product data coming from several heterogeneous design sources and provides a single source of information to support a suite of related engineering analyses. This representation - named *Analysis-Oriented Product Model* (AOPM) - eventually evolved into the Analyzable Product Model (APM) presented in this thesis. The AOPM was defined as an abstracted *view* of the design-oriented product data that is more "appropriate" for engineering analysis in that:

- It contains entities whose names, attributes and structure are more suitable for use by analysis models;
- It contains mostly data that is used by the analysis models, which is a subset of all the data generated by the design tools; and
- More importantly, it supports idealizations of the design data that can be shared by multiple analysis models.

The utilization of an AOPM - and the technique used to populate it - were demonstrated in this paper with a simple test case involving thermomechanical analysis of electrical

components. As shown in Figure 7-2, this test case consisted of a simple component extensional analysis performed with data coming from two hypothetical applications: an E/CAD application, used to define electrical components and their geometry, and a Material Definition application, used to populate a database of material properties. The purpose of the analysis was to determine the change in length of an electrical component due to a change in temperature.

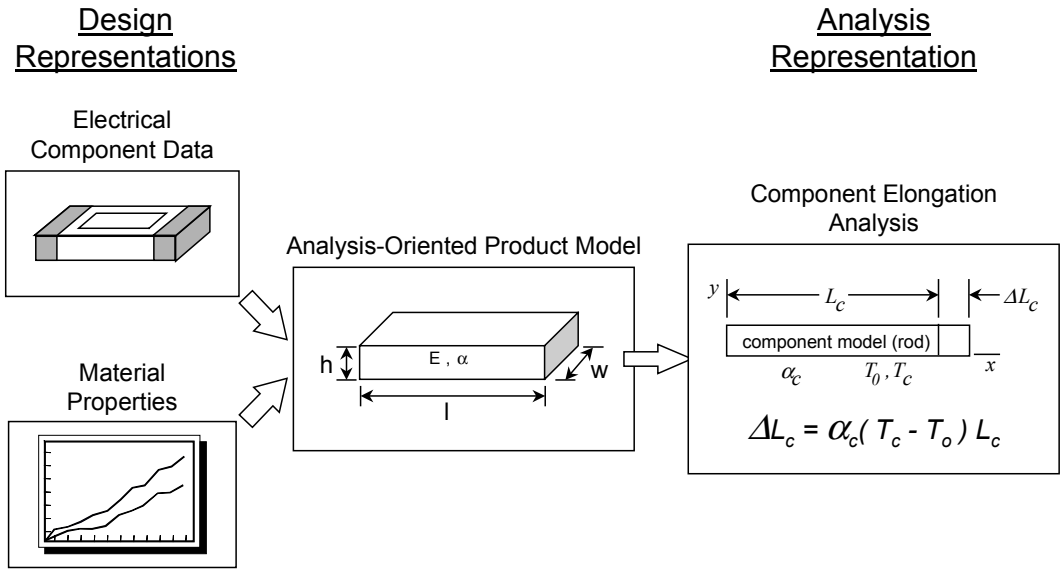


Figure 7-2: Component Extensional Analysis Test Case

The AOPM of this case was defined using the EXPRESS data modeling language (Appendix A). The corresponding (partial) EXPRESS-G diagram is shown in Figure 7-3. This AOPM defined entities such as resistors, integrated circuits, and electrical packages as well as their attributes (such as the electrical component's part number and the resistor's base material). It also included idealized attributes required by the component elongation analysis such as the electrical component's primary structural material and the electrical package's bounding box length (indicated with asterisks in the EXPRESS-G diagram). The operations needed to calculate the values of these idealized attributes were defined as part of the definition of the AOPM as EXPRESS WHERE rules.

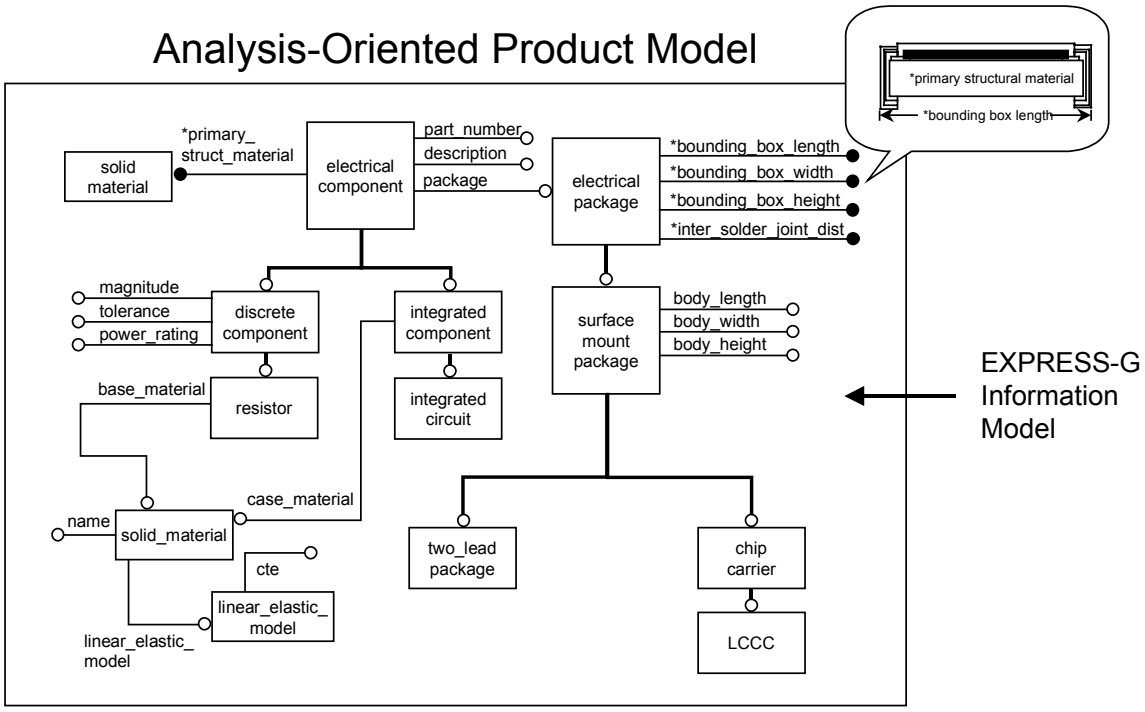


Figure 7-3: AOPM for the Component Extensional Analysis Test Case (partial)

Next, the mappings between the two design representations and the AOPM were defined. These mappings, shown graphically in Figure 7-4, define how the values of the attributes in the AOPM are computed from values in the two design representations. Idealized attributes are left empty during this mapping, as they will be calculated on demand when they are required by the analysis.

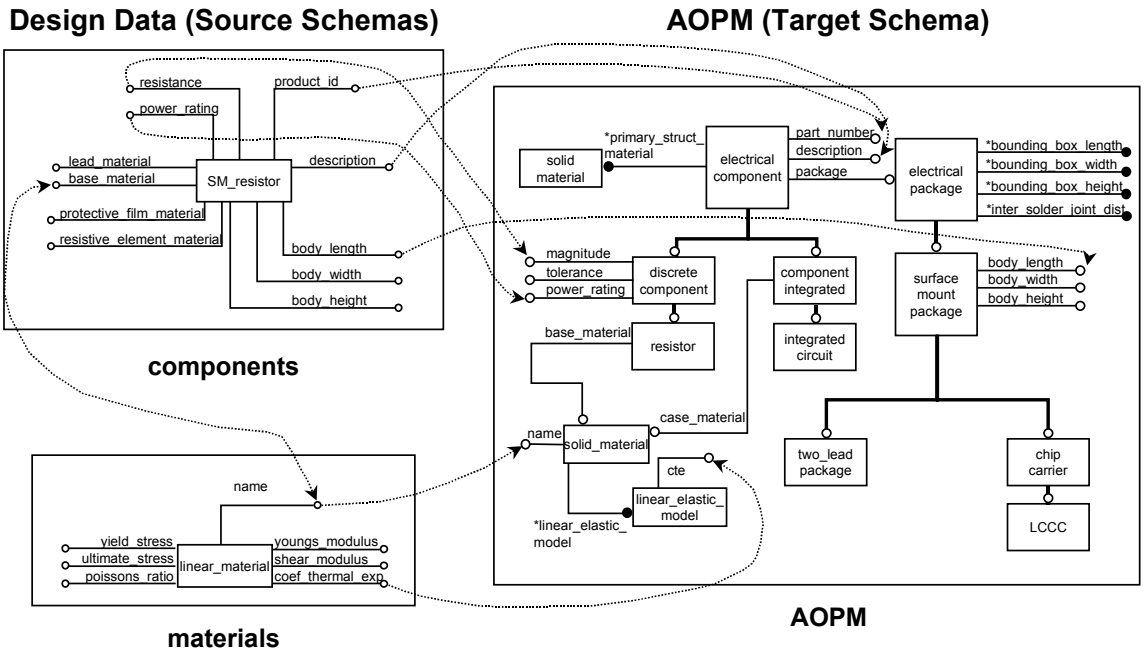


Figure 7-4: Mappings between the Design Representations and the AOPM

Figure 7-5 illustrates how the information in the AOPM is linked to the analysis variables of a particular type of analysis model (that is, how the analysis model *uses* the AOPM). For example, the bounding box length attribute (an idealized attribute of the AOPM) is linked to the analysis variable L of the elementary rod analysis model (linkage Φ_2), and the coefficient of thermal expansion of the primary structural material (another idealized attribute) is linked to the analysis variable α (linkage Φ_1).

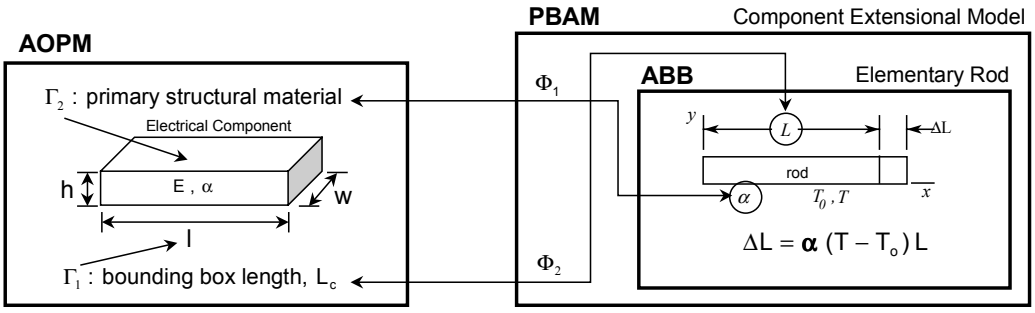


Figure 7-5: AOPM-Analysis Model Linkage

The EXPRESS definitions of the entities in the AOPM and the idealizations they support were implemented in C++ so that they could be used in the development of the analysis application. As illustrated in Figure 7-6, each entity in the AOPM was implemented as a C++ class, and the attributes of this entity as class variables of this class. The protocol of this class consists of member functions to access and update the values of the attributes of the entity as well as member functions that implement the WHERE rules and allow access to the values of the idealized attributes.

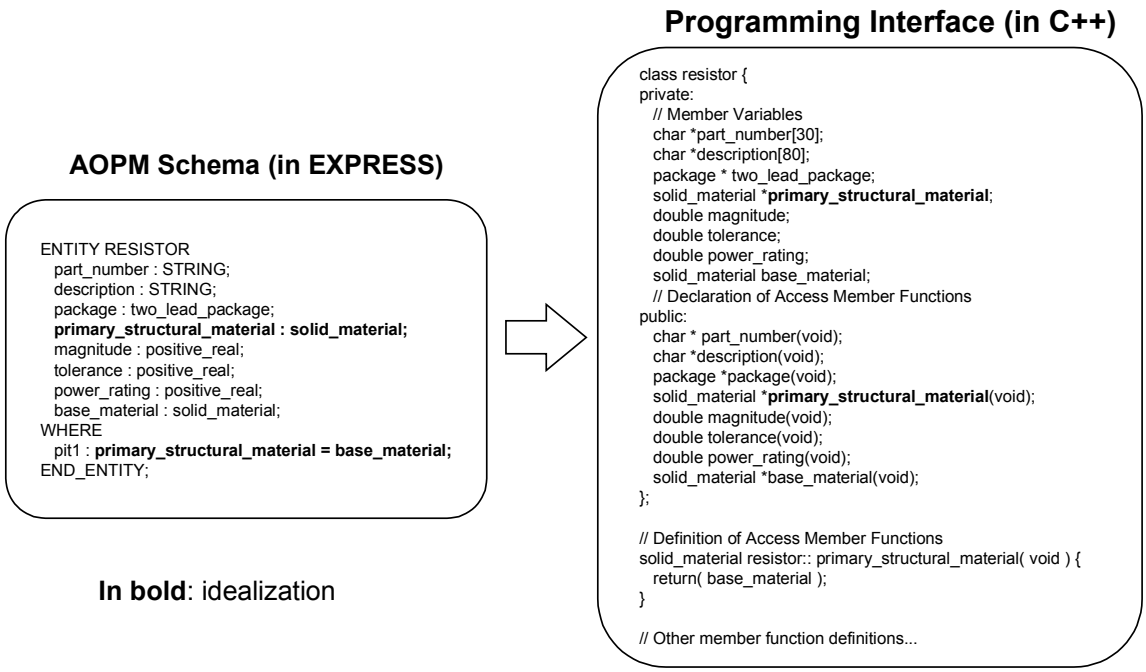


Figure 7-6: AOPM Implementation

For this test case, the analysis application was a simple C++ program that implemented a formula-based extensional model using the classes defined in the AOPM. Figure 7-7 is a screen shot of this program displaying the total strain and elongation values obtained for a particular resistor and temperature variation.

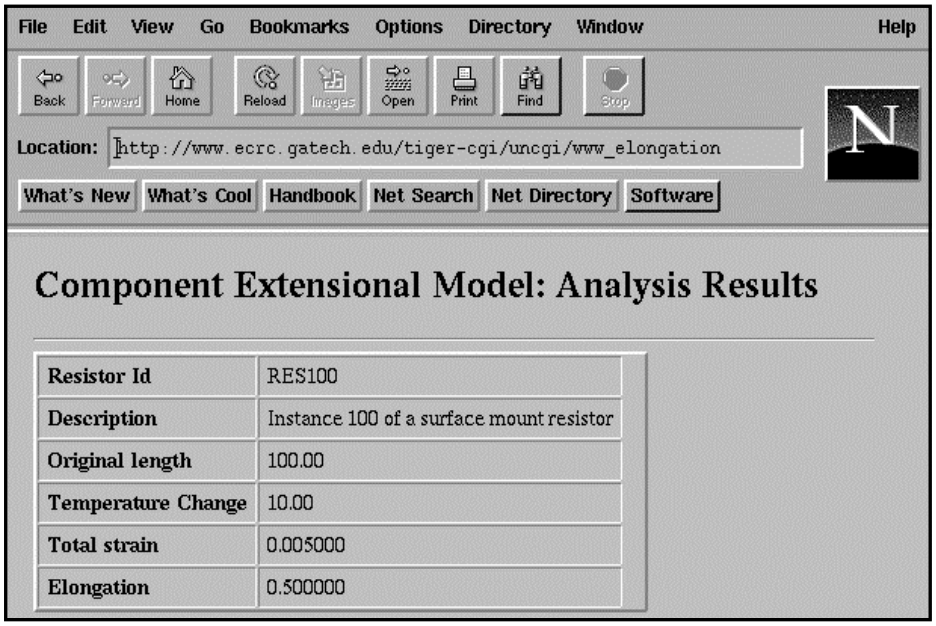


Figure 7-7: Component Extensional Analysis Application

The overall data flow of this test case is shown in Figure 7-8. This diagram shows how the data generated by each of the design applications is translated into STEP, mapped into the AOPM and, finally, used by the analysis application.

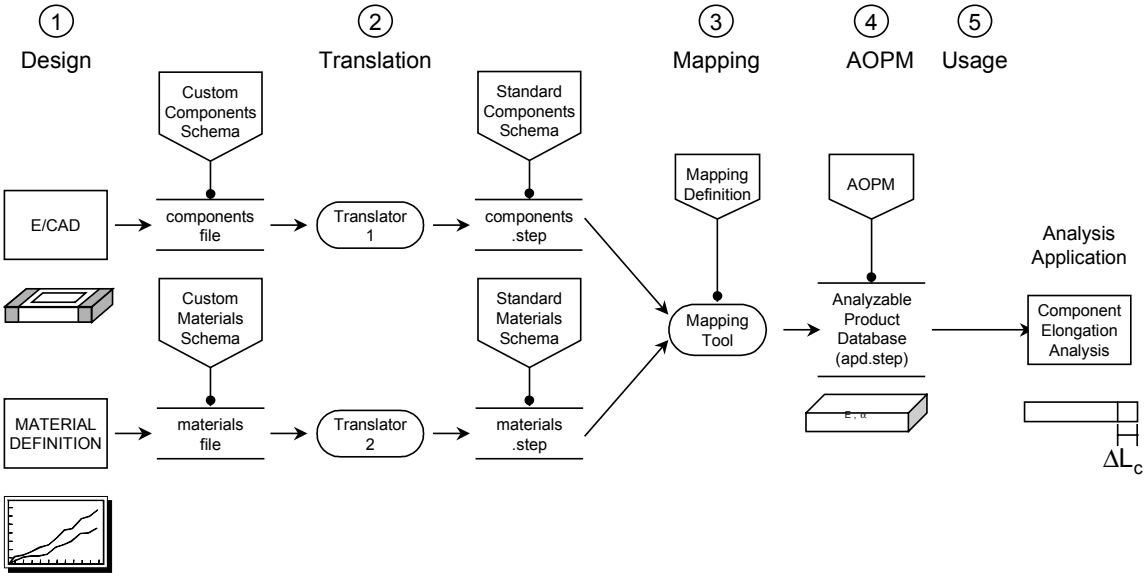


Figure 7-8: Overall Data Flow of the Component Extensional Analysis Test Case

Even though the structure of the data, idealizations and analysis model of this test case were relatively simple, this test case helped introduce and test the preliminary AOPM ideas: how to populate an AOPM, how to support idealizations and how an AOPM is used by analysis applications. The next step was to test these concepts in a more realistic scenario, using a STEP Application Protocol as the schema for the design data, more complex idealizations and more realistic design and analysis applications. This was accomplished during the DARPA-sponsored project TIGER (Team Integrated Electronic Response) (EIS Lab 1997; Peak, Fulton et al. 1997; Scholand, Peak et al. 1997; SCRA 1997; Tamburini, Peak et al. 1997) described next.

Team Integrated Electronic Response (TIGER) Project

The goal of the TIGER project was to demonstrate a collaborative design and manufacturing scenario in which a small manufacturing enterprise (SME) exchanged design information with the prime contractor early in the design process, thus reducing the iterations necessary to produce a successful design. For this purpose, a suite of design, manufacturing, and communications tools integrated across the Internet was made available to the product development team. The domain demonstrated was the design, fabrication, and assembly of printed wiring boards (PWBs) and printed wiring assemblies (PWAs).

In the TIGER scenario, a PWA designer generated PWA/B design information and sent it to a PWB manufacturer in STEP AP210 format. When the PWB manufacturer received this file, he uploaded it to an Internet-based engineering service bureau over the Internet (Scholand et al. 1997) that provided a variety of design and analysis services including design-for-manufacturability (DFM) and thermomechanical analysis. These services were integrated in an analysis environment developed for TIGER called *DaiTools-PWA/B* (Peak et al. 1997). Once the AP210 file was uploaded to the engineering service bureau, the PWB manufacturer invoked - from *DaiTools*' interface - a tool called PWB Layup Design Tool (Figure 7-9). He used this tool to specify the detailed layup of the PWB by selecting specific laminates, prepregs, and copper foils that physically realized the requirements specified by the PWA designer in the AP210 model.

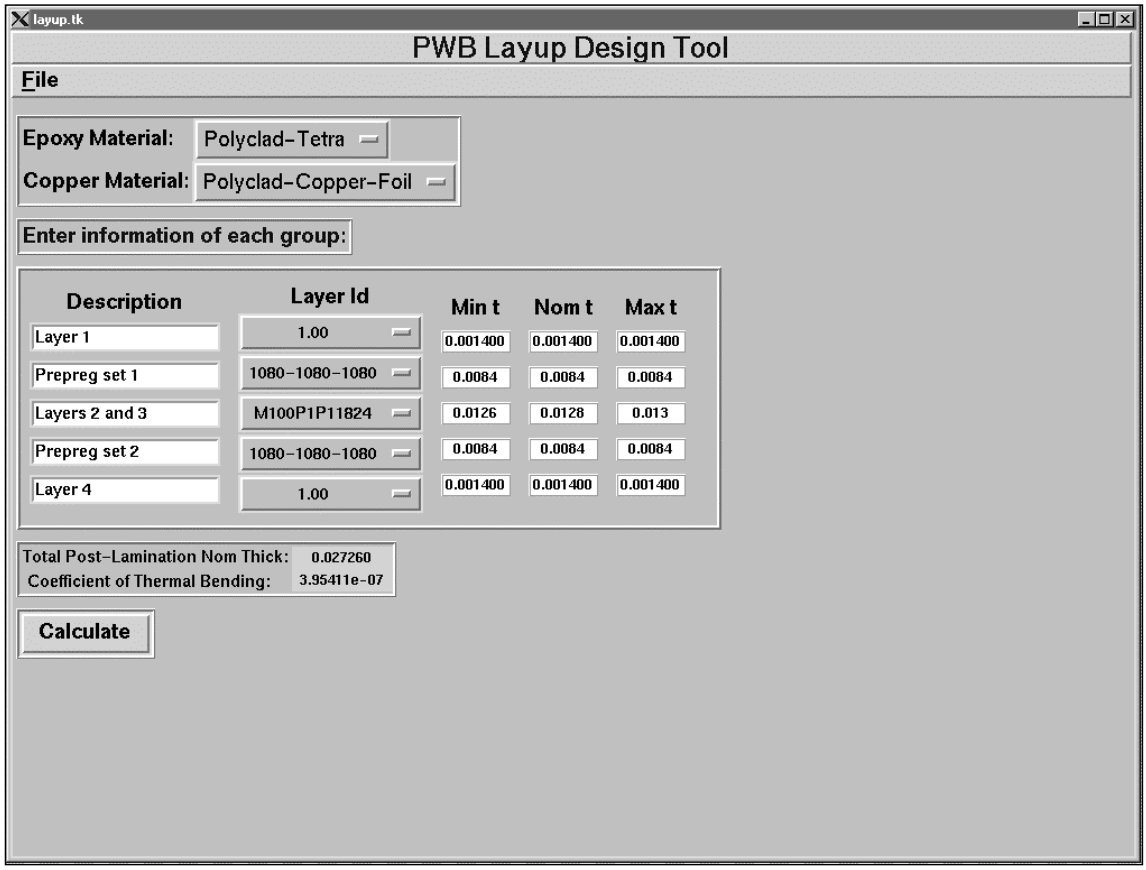


Figure 7-9: TIGER PWB Layup Design Tool

As these layup details affect PWB thermomechanical behavior, the PWB manufacturer had to perform some analyses to check the impact of his decisions. For this reason, he invoked the Warpage Analysis Application (Figure 7-10) to assess the warpage undergone by the board due to changes in temperature that occur during manufacturing. Two fidelities of warpage analysis detail could be requested: a quick formula-based warpage analysis and an FEA-based plane strain warpage analysis. The PWB manufacturer performed this design-analysis iteration until he was satisfied with the layup. Other analyses modules offered by *DaiTools* were a PWA deformation analysis (to asses the warpage of the board with the components on it), a solder-joint deformation and fatigue analysis (to assess joint deformation and fatigue life due to temperature changes on a component basis), and a plated-through hole deformation module (to assess deformation inside plated-through holes due to changes in temperature).

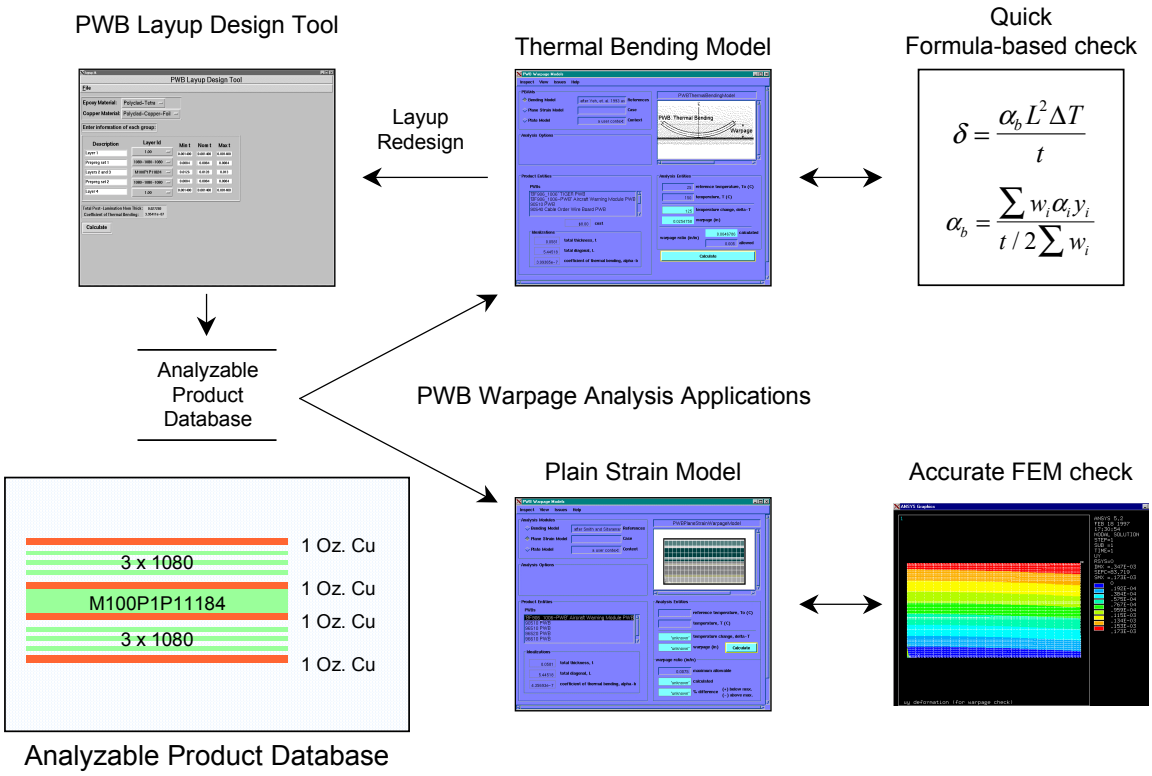


Figure 7-10: PWB Layup Design and Analysis Cycle

The Analyzable Product Model (APM, Figure 7-11) provided the integration of information needed to drive this design-analysis process. When the AP210 file was uploaded to the engineering service bureau, *DaiTools* read it in and combined it with other information to form the Analyzable Product Database (APD). The APD became the only source of information required to support the analyses offered in *DaiTools*.

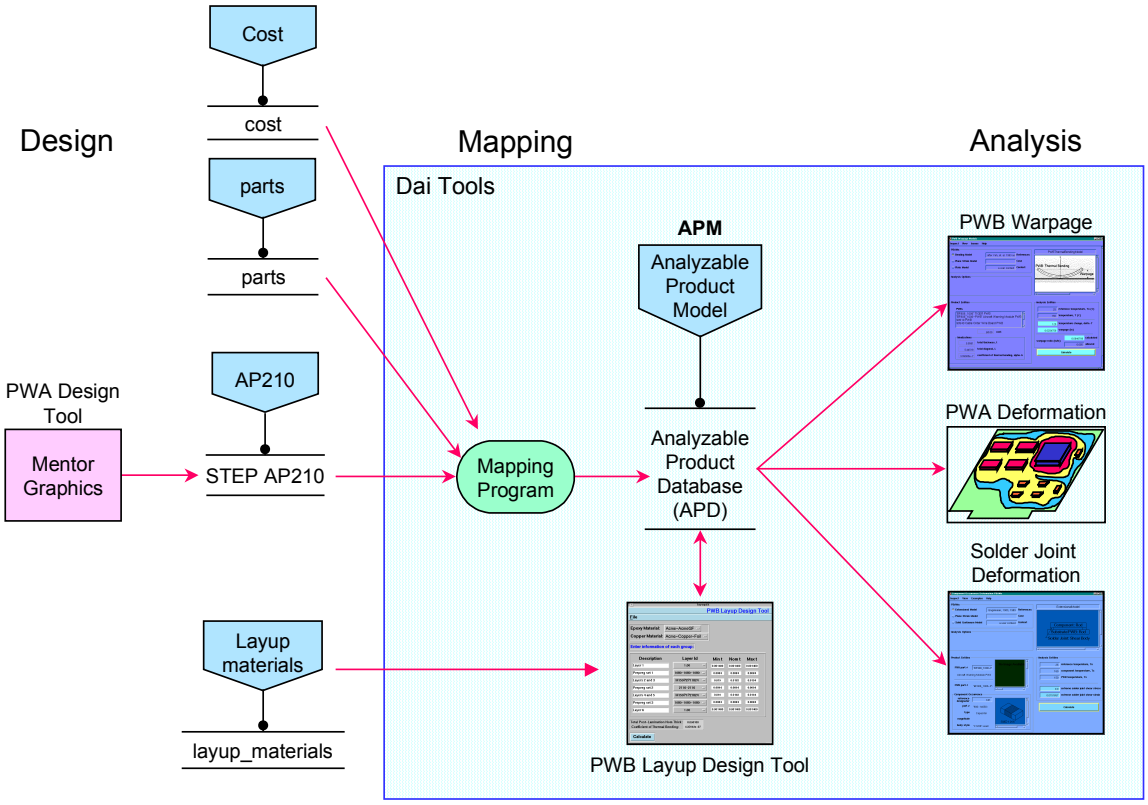


Figure 7-11: TIGER Data Flow

In order to support the design-analysis scenario demonstrated in TIGER, the basic AOPM developed for the Component Extensional Analysis test case discussed above had to be significantly extended with new entities and idealizations. The analyses supported by TIGER required more information about the electrical components (geometry, package types, placement on the board, material) and the board (detailed layup, materials, geometry). The new idealizations implemented were the total diagonal of the board, total thickness of the board and coefficient of thermal bending (α_B) of the board. The first idealization (total diagonal of the board) is computed considering an imaginary bounding rectangle surrounding the outline of the board, and assuming the length of the diagonal of this rectangle as the total diagonal of the board. The second idealization (total thickness of the board) requires the detailed layup of the board to calculate the post-lamination thickness, which takes into account the flow of epoxy material between the traces of the conductive layers when the board is heated and subjected to pressure during lamination. Finally, the

coefficient of thermal bending is a lumped material property of the total layup and is calculated as a weighted sum of individual stratum properties.

The work done for the TIGER project helped to further develop the APM concept and test the initial ideas with a realistic design-analysis example. The variety and complexity of the analyses supported helped to gain more understanding about the general requirements of an APM. Also, the utilization of a real STEP Application Protocol (AP210) as the source of design information raised several fundamental problems such as how to add missing information not contained in an AP but required for analysis, and how to coordinate and map information spanning several STEP repositories. It also added the critical link to commercial design tools such as Mentor Graphics.

The AOPM/APM work done until TIGER had one significant drawback: the code to access the analyzable product model and the idealizations defined in it was specific to the test case, in other words, it was *early-bound*. Almost none of the prototype code done for TIGER could have been reused for a test case involving, for example, aircraft structures. In addition, idealizations were implemented manually by directly modifying the access methods of the classes corresponding to entities in the product model. For example, recall the C++ example from TIGER of Figure 7-6, in which the idealization to obtain the primary structural material of a resistor is manually implemented as an access method of the resistor class. Also, with this approach, a class method had to be written for each expected input/output combination.

Product Simulation Integration (PSI) Structures Project

By the time of this writing, the concepts and techniques demonstrated in TIGER are being extended and applied towards the Product Simulation Integration (PSI) Structures Project. The PSI project is a multi-team, multi-year project conducted by The Boeing Commercial Airplane Group in Seattle, Washington. The objective of the PSI project is to define and enhance the processes, methods and tools to integrate structural product simulation and analysis with structural product definition (Prather and Amador 1997). This includes automated engineering analysis as an integral component of the product definition. The EIS Lab team has been contributing to this effort since September of 1997 with the application of its MRA/APM techniques in the airframe structures, extending beyond the electronics

domain explored in earlier work. Specifically, it is providing design-analysis associativity techniques that are crucial to the true simulation integration the project wants to achieve (Peak, Fulton et al. 1999).

Product Data-Driven Analysis in a Missile Supply Chain (ProAM) Project

Another important design-analysis integration project in which the EIS Lab team is currently participating is the Product Data-Driven Analysis in a Missile Supply Chain (ProAM) project (EIS Lab 1998), also being conducted by members of the EIS Lab. As in TIGER, ProAM's goal is to demonstrate a collaborative design and manufacturing scenario between SMEs and prime contractors in which SMEs access advanced analysis capabilities through an Internet-based engineering service bureau (ESB) and provide feedback early in the product development cycle. Also as in TIGER, the representative test case chosen for ProAM is the thermomechanical analysis of PWAs and PWBs. The prime contractor in the ProAM demonstration scenario is the Aviation and Missile Command's (AMCOM) Manufacturing Science and Technology (MS&T) Division, and the SMEs are small PWB manufacturers. The ProAM project is also providing significant input to the GenCAM standard, an IPC (Institute for Interconnecting and Packaging Electronic Circuits) data transfer standard being documented in a series of standards identified as IPC-2510 (Institute for Interconnecting and Packaging Electronic Circuits 1999a; Institute for Interconnecting and Packaging Electronic Circuits 1999b). This standard specifies data file formats used to describe printed board and printed board assembly products with details sufficient for tooling, manufacturing, assembly, inspection and testing requirements. These formats may be used for transmitting information between a printed board designer and a manufacturing or assembly facility. The files are also useful when the manufacturing cycle includes computer-aided processes and numerical control machines.

Other Design-Analysis Integration Research

Modeling Semantic Integrity in Design-Analysis Information Flows

Eastman (1996) introduces a new representation for modeling semantic integrity in engineering design. In this representation, the design and analysis of an engineering product is modeled as a network of design/analysis operations of the form:

$$\Phi_i = (\{E\}_i^R, \{E\}_i^W, \{C\}_i^B, \{C\}_i^A)$$

Where:

- $\{E\}_i^R$ (readset entities) is the set of classes read as input to application Φ_i;
- $\{E\}_i^W$ (writeset entities) is the set of classes written as output upon successful completion of application Φ_i;
- $\{C\}_i^B$ (before-constraint set) is the set of integrity constraints to be evaluated before the execution of application Φ_i. The scope of $\{C\}_i^B$ - that is, the classes accessed in its evaluation – includes $\{E\}_i^R$;
- $\{C\}_i^A$ (after-constraint set) the set of integrity constraints to be evaluated after the execution of application Φ_i. The scope of $\{C\}_i^A$ includes $\{E\}_i^R \bigcup \{E\}_i^W$;

Readset entities are the classes read by the application, whereas writeset entities are the classes modified in the application, which may overlap with the readset classes. Associated with an entity class E_i is a set of constraints (or integrity rules) denoted $\{C_{E_i}\}$. The constraints $\{C_{E_i}\}$ are inherited into the set of all instances of class E_i. Constraints may or may not have a function body. Those without a function body serve as shadows for an external application and their state is treated as a flag by the application interface, which sets the constraints instance states corresponding to the operation taken. Those that have a function body are executable and derive the constraint's state when applied to its arguments.

Since design applications may be executed multiple times, an operation instance – denoted ϕ_{ij} - is defined which is an instance of application Φ_i as:

$$\phi_{ij} = (\{e\}_{ij}^{R}, \{e\}_{ij}^{W}, \{c\}_{ij}^{B}, \{c\}_{ij}^{A})$$

Where:

$\{e\}_{ij}^{R}$ are instances of $\{E\}_{i}^{R}$;

$\{e\}_{ij}^{W}$ are instances of $\{E\}_{i}^{W}$;

$\{c\}_{ij}^{B}$ are instances of $\{C\}_{i}^{B}$. Their possible values are **`True`**, **`False`**, **`Undefined`** and **`NULL`**;

$\{c\}_{ij}^{A}$ are instances of $\{C\}_{i}^{A}$.

When an operation instance ϕ_{ij} is performed, new entity instances are created, deleted or modified. In addition, the state of constraint instances changes. It is both the entity instances and the constraint instances that determine the state of the design model and manage the communication between one operation instance and others. Constraints are satisfied incrementally by a sequence of operation instances; each operation instance adding to the set of integrity rules already satisfied, thus incrementally building up the design model. Design is considered completed when a state of total integrity is achieved for all instantiated constraints.

Eastman's representation also supports the case in which multiple constraints are associated with a single design variable. In such cases, as he points out, it is likely that some new operation instance will modify a variable after it has been set to satisfy other design constraints. In order to maintain the design model in a valid state an operation instance that modifies the variables accessed by a constraint instance must set the constraint instance to **`NULL`**, forcing the re-evaluation of all other constraint instances whose parameter values have changed.

Eastman also explains that many existing engineering and manufacturing applications, many of which have long-standing use and validation, do not have to be rewritten in order to accommodate his representation. Instead, they have surrounding or "wrapping" code that performs the necessary translations and serves the purpose and has the general form of the

equation for ϕ_{ij} shown above. Prior to extracting readset data, the wrapper checks the value of the before-constraints to determine if the application can be executed. If so, the data is extracted and the operation is executed. A successful operation instance results in both new or modified data being assigned and the after-constraints set to `True` for the instances being written. As new data are written, the effects of the changes are propagated. The changed constraint flags are then available to be read by the before-constraints of other applications.

Eastman, et al. (1995; 1995; 1991; 1994) developed the Engineering Data Model (EDM) and its database implementation (EDM-2) as a platform for both representing design information and supporting translation between different applications views. EDM is a data model tailored for product modeling that consists of both a graphical notation and a textual definition language. EDM-2 is a database management system based on EDM whose intended use is for the implementation of back-end databases supporting the integration of a heterogeneous and evolving set of design applications. It incorporated operations for data management and is not meant to support design operations directly, as these operations are the responsibility of external applications. As a back-end database, it addresses the following capabilities:

- Translation of data between the database views corresponding to different application interfaces (Assal and Eastman 1995). EDM makes translation a task of the database itself, and in order to do this it defines some structures (design entities, constraints and maps) that capture the relationships of the object types and provides mechanisms for managing the integrity of the views when they are updated, possibly in an arbitrary order. It also provides a mechanism for deriving dependent data and generating and maintaining equivalent views. This mechanism allows storing different representations of the same product in a unified database and provides means for translating from one representation to the other.
- Managing the integrity of data, especially among concurrent users making iterated decisions (Eastman, Cho et al. 1995).
- Version control and iteration to earlier design stages.
- Dynamic model evolution, in support for new applications, as needed both during design and over the product life cycle (Eastman, Assal et al. 1995). EDM-2 defines

distinct constructs for supporting model evolution, which allow extensions to be made to both the model schema and object instances.

The approach of defining views of product data to support specific applications is similar to the one presented in this thesis. This thesis is, however, more specific in that it focuses on the definition of *analysis-oriented* views of design data aimed at supporting the needs of *analysis* applications. Despite this difference in scope, there are some interesting similarities between the EDM language and the APM-S language presented in this thesis (Subsection 52). For example, EDM defines constructs called *Design Entities*, which are similar to APM-S *domains* (they both define classes and their attributes). Likewise, the instance definition mechanism of EDM is similar in scope to the instance definition language presented in this thesis APM-I (Subsection 53). EDM *constraints* are similar to APM-S *relations* in that they both define relations among attributes. The difference is that in EDM the main purpose of constraints is to ensure data integrity, whereas in APM-S relations are used to define bi-directional mappings between design and analysis representations (in the case of product idealization relations), or to derive the values of redundant product attributes (in the case of product relations). In this sense, APM-S relations are more similar to EDM *maps*, which are specializations of constraints used to translate data from one representation to another. However, unlike APM-S relations, maps define the data translation in one direction only. For example, a map can be defined to translate an IGES line into a DXF line. If the opposite mapping is needed, a separate map must be defined. Another difference is in how constraints and relations are defined and implemented. In EDM, constraints and maps are defined as calls to external functions implemented in some target programming language and dynamically linked into the database. In APM-S, relations are fully defined as part of the APM definition and resolved at run time by an external constraint solver.

Graph Grammar-Based Representation Conversion

Rosen, et al. (1992; 1991; 1994), addressed the problem of integration of CIM (Computer Integrated Manufacturing) functions through viewpoint-specific feature-based representations. They describe the use of formal graph grammars (a generalization of string grammars) to define two representations: a feature-based design representation of thin-walled components and a manufacturing representation. The first representation captures the features and their geometry and adjacencies of components that can be manufactured by

injection molding, die casting, and sheet metal stamping processes, whereas the second representation captures manufacturing features and cost drivers. Next, they present a general conversion methodology to convert the feature-based design representation into the manufacturing representation. The resulting manufacturing representation is then used to perform tool construction cost evaluations. They illustrate their conversion methodology by converting the feature-based design representation of a simple thin-walled component, which is to be injection-molded, into its corresponding manufacturing representation and performing a tool cost evaluation based on the resulting manufacturing representation. Although Rosen's conversion methodology was not specifically developed for design-analysis integration and focuses on feature-based representations, its general purpose – to convert information between two representations that have been formally defined with some kind of formal language – is very similar to the one of the APM Representation presented in this thesis.

Design Idealization using Artificial Intelligence Techniques

Shephard, et al. (1992) describe how physical descriptions of multichip modules (MCMs) are converted into idealized representations that are then used to perform thermal and thermomechanical finite-element analyses. The physical description of the MCM is considered as the driving representation of all the subsequent analysis steps in the process. This physical description is composed of two parts: the first part consists of the geometric model information and the second part of non-geometric information (or "attributes") such as material properties, environmental conditions and boundary conditions, required to complete the physical description. Analysis idealizations processes then use a set of "interrogation functions" to obtain information not inherently in the model's data file and convert the physical description into the idealized representation finally analyzed. For the test case presented, the source of the physical description of the MCM is a CIF (Mead and Conway 1980) file[3]. CIF files alone do not contain enough information to drive the idealization process required to perform the thermal and thermomechanical analyses. Thus, the approach taken was to supplement the CIF file with the additional information needed to complete the physical specification of the MCM. An idealization control system called IDEALZ, developed by the authors, provides explicit control of the idealization steps used

[3] CIF stands for CalTech Intermediate Form, a graphics language which can be used to describe integrated circuit layouts.

during engineering design and the coordination required in the design process. IDEALZ, described in detail in (Shephard, Baehmann et al. 1990; Shephard, Korngold et al. 1990; Shephard and Wentorf 1994), uses AI techniques to guide the user through the design process by interpreting his analysis requests, determining if they are reasonable, developing an idealization strategy, interacting with the various modelers and applications to get the analysis results and adaptively refining the idealization until the desired level of accuracy is achieved. A graphical user interface for IDEALZ is described in (Wentorf and Shephard 1993).

Finn, et al. (1993; 1992) also utilize AI techniques to obtain an idealized representation of the product - more suitable for analysis - from a CAD representation. They describe an interactive modeling system for assisting engineers in the process of selecting, applying and evaluating candidate mathematical modeling options in order to obtain the idealized model for analysis. An engineer using this system first constructs the design problem with the help of a CAD system and a case base of modeling options. The CAD system allows the engineer to specify the geometric features of the physical system, while the case base of modeling options allows him to specify the phenomena and boundary conditions. Once the user constructs the problem, the modeling assistant creates a knowledge-based CAD representation of the problem which forms the basis for matching and retrieving suitable base cases. The engineer then selects a particular modeling option and the system automatically evaluates the problem by applying the appropriate engineering formulae and solving them. New candidate models can be assessed by adding or removing features, specifying alternate phenomena or boundary conditions, reducing dimensions, taking symmetries or substituting material models. The results allow the engineer to compare different candidate models and assess the effect of particular modeling decisions.

Design-Analysis Integration for Finite Element Analysis

Arabshahi, et al. (1991; 1993) point out the fact that although complete geometric information for the product is often available in the form of a solid model, this is rarely taken advantage of due to the amount of time required to simplify and idealize the geometry for the subsequent meshing stage. For this reason, they say, analysts often find it easier to reconstruct the idealized model from scratch, a process which is error prone and prohibits linking the analysis results to the product in a formal way. The unfortunate result is that a

large proportion of the analyst's time is spent preparing an idealized model of a product for analysis. They describe a system which would allow a more automated transition from a solid model to an idealized model suitable for finite-element analysis. The functional components of such a system are: 1) A Product Description System (PDS) in which reside both the geometry and non-geometric attributes such as environment conditions, design requirements, manufacturing method and costs, 2) An intelligent, semi-automated means for transforming the geometric and non-geometric data stored in the PDS into an attributed, abstracted model suitable for finite element mesh generation, 3) Intelligent meshing routines with varying degrees of automation to suit the application, 4) Finite element solvers to suit the range of analysis problems, and 5) Post-processing including the ability to associate results with the idealized model to allow for modifications to this model (adaptive idealization).

Agent-Based Engineering Tool Integration

Cutkosky, et al. (1993) describe a demonstration project called The Palo Alto Collaborative Testbed (PACT) whose goal is to develop an infrastructure which integrates multiple sites, subsystems, and disciplines to facilitate concurrent engineering. PACT's architecture is based on programs that encapsulate engineering tools called *agents*. Communication among agents is achieved by standardizing: a) the services that agents may request of one another, b) how knowledge (constraints, negations, disjunctions and rules) is exchanged among agents, and c) the vocabulary (classes, relations, functions, and object constants) shared among agents, also known as *ontology*. Communication between applications is achieved exclusively through their corresponding agents. In contrast with traditional product data exchange approaches, PACT utilizes no shared models at all because, as the authors argue, it is a problem for different design tools to share the same model. They also add that a single shared database encompassing all the data of participating tools would quickly become a bottleneck. However, the authors admit that setting up the communication framework between agents requires too much interaction and negotiation between the developers of the tools in order to agree on a shared ontology. They mention that one possible solution to this problem is to take advantage of the standardization efforts made by the STEP community to define the necessary ontologies. Since STEP is a formal standardization effort, any necessary agreement is handled by the developers of the standard, not by individual tool developers.

Multi-Model Design-Analysis Integration

Sahu and Grosse (1994) describe a three-model representation to enable integration between design and analysis modules. The *primary representation* of a design consists of a high level representation of its geometry, such as that contained by a feature-based solid model, and associated intent-specific information concerning both the design intent and its manufacturing process. The *secondary representation* provides a representation of the model suitable for numerical solvers; a common form of secondary representation is a finite element model of the design's primary representation. The information stored in the primary representation helps determining and imposing the boundary conditions, material properties and loading conditions. The raw data from the numerical solution (that is, the finite element solution) constitutes the *tertiary representation*. The tertiary representation is the lowest possible representation of the design and has little meaning until it is associated with the secondary representation and transformed into a qualitative description useful for design modification. The authors describe and implementation of this methodology, a system called Cognitive Symbolic and Numeric Designer (CSN-Designer). CSN-Designer assists the designer in making intelligent design changes based on functional and manufacturing analyses. The authors argue that the up-front use of analysis and manufacturing simulation results can provide guidance to the design engineer during the early stages of design, and that this has been the primary motivation in building coupled systems for design and analysis tools. They call this concept "analysis *for* design".

Mathematical Modeling and Simulation Languages

A significant multinational research effort – coordinated by the Federation of European Simulation Societies (EUROSIM) - is currently taking place in Europe to develop a mathematical modeling and simulation language called Modelica (Elmqvist and Mattsson 1997; Elmqvist, Mattsson et al. 1998a; Elmqvist, Mattsson et al. 1998b; Fritzson and Engelson 1998; Mattsson and Elmqvist 1998). The aim of Modelica is to unify the concepts from several modeling languages available from universities and small companies into a common basic syntax and semantics and to design a new unified modeling language. The main objective is to make it easy to exchange simulation models and model libraries and to allow users to benefit from the advances in object-oriented modeling methodology. Modelica builds on non-causal modeling with true equations and the use of object-oriented

constructs to facilitate reuse of modeling knowledge. Modelica is intended for modeling within many application domains (electrical circuits, multi-body systems, drive trains, hydraulics, thermodynamical systems, chemical systems, etc.) and possibly using several formalisms (ordinary differential equations, differential-algebraic equations, bond graphs, finite state automata, Petri nets, etc.). Tools that might be general purpose or specialized to certain formalism and/or domain will store the models in Modelica format in order to allow exchange of models between tools and between users, thus promoting reuse. The main features that distinguish Modelica from other modeling languages are:

- *Object-oriented modeling:* this techniques makes it possible to create physically relevant and easy-to-use components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- *Non-causal modeling:* modeling is based on equations instead of assignment statements as in traditional input/output abstractions. Equations do not specify which variables are inputs and which are outputs, whereas in assignment statements variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equations-based models is unspecified and fixed only when the equation systems are solved (this is called non-causal modeling). Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used (in other words, they can be used with multiple input/output combinations of data). This generalization enables both simpler models and more efficient simulation.
- *Physical modeling of multiple domains:* model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to signal blocks. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

Modelica programs are built from *classes*. Like in other object-oriented languages, classes contain variables, that is, attributes representing data. The main difference compared with traditional object-oriented languages is that instead of functions (class methods) Modelica

uses *equations* to specify behavior. For example, the following Modelica construct defines a resistor and the equations that relate its resistance, voltage and current:

```
class Resistor "Ideal electrical Resistor"
 extends TwoPin;
 parameter Resistance R(unit="Ohm") "Resistance";
equation
 R*i = v;
end Resistor;
```

Where `TwoPin` is defined as (`Pin` and `Voltage` are defined elsewhere):

```
partial class TwoPin "Superclass of elements with two pins"
 Pin p, n;
 Voltage v;
 Current I;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

Classes, instances and equations are translated into flat set of equations, constants and variables. After flattening, all the equations are sorted, simplified and then converted to assignments statements by a symbolic solver. Finally, C, C++ or Java code is generated, and it is linked with a numeric solver. The initial values can be specified by the user as part of the Modelica definition, by means of the `parameter` keyword.

The Modelica language has significant similarities with the APM-S language introduced in this thesis (Subsection 52). Both languages are used for defining object-oriented, reusable modeling components and model the equations relating their attributes in a non-causal way. Symbolic solvers are also used in both works to solve for the values of unknown attributes. The authors of Modelica are also developing a library of the most commonly used components that can be shared between applications, similar to the library of ABBs introduced by Peak (Subsection 9). What differentiates the research presented in this thesis from the Modelica research is that this thesis focuses on modeling *products* for analysis, whereas Modelica applications have focused on modeling analysis models (although it is likely that the concepts could easily be applied to product modeling for analysis). In addition, this thesis makes particular emphasis on multi-directional, multi-fidelity, reusable idealizations, and elaborates more on how these analyzable product models are populated

with information coming from multiple design sources and how they are connected to multiple analysis models.

Design-Analysis Integration using STEP

Until recently, the focus of the STEP standard has been on the exchange of product data between applications with similar scope and semantics. For this reason, the standard had not specifically addressed the problem of integrating design and analysis applications. Currently, however, there is a growing interest from the STEP community on improving the standard to support analysis information and facilitate design-analysis integration. This section overviews some past works involving STEP that are relevant to the topic of design-analysis integration including the Engineering Analysis Core Application Resource Model (EA C-ARM), the ongoing ISO standardization effort aimed at creating a standard representation for engineering analysis information.

AP210-Driven PWA Fatigue Analysis

Rassaian, et al. (1997; 1995) describe what is perhaps the first commercial utilization of a STEP Application Protocol (AP210) as the input for an engineering analysis. They describe a system which utilizes a STEP translator to extract data from a CAD Printed Wiring Assembly (PWA) database, automatically builds finite element structural and thermal models from the design data, and performs structural and thermal analyses. The results of these analyses - together with data from component, material and environment databases - are then imported to an in-house-developed fatigue analysis code called Fatigue Synthesis for Avionics Programs (FSAP), which helps the user predict the fatigue life of every part on a PWA and provides a series of options to solve any problem encountered. For the calculation of fatigue life, FSAP provides closed-form algorithms (for pre-built analysis models, each representing a specific package type) as well as finite element analysis (for custom package types that are not in the library of analysis models).

Defining Product Views using STEP Mapping Languages

Hardwick (1994) describes how a STEP mapping language (EXPRESS-V – Appendix A) is used to define views of an integrated database and to integrate the applications used in an automotive company. This integrated database is formed by the union of overlapping subsets (known as Application Interpreted Constructs or AICs) of two STEP application protocols: AP203 (which supports geometric descriptions) and AP207 (which supports sheet metal die descriptions). A department of an automotive company receives AP203 descriptions for a car body and issues AP207 sheet metal die descriptions that contain the process plans, material descriptions and change orders that an automotive supplier needs to make dies for the sheet metal parts of the body. EXPRESS-V is used to define the bi-directional mappings between the integrated database and each of the AP views. These views support the information needs of different applications including a geometric modeler, a materials system, a process planning system and a project management system. Although this paper does not explicitly address the problem of integrating design and analysis applications, application-specific views could potentially be defined to support the needs of analysis applications.

A similar data exchange approach based on EXPRESS mapping languages was developed by Gadient and Hines (1994). In their paper, the authors describe an application of STEP mappings in their EXPRESS-Driven Data Conversion (EDDC) architecture. They implement an EDDC to convert electrical product data (printed circuit assemblies) from Mentor Graphic's Board Station to STEP AP210. In the EDDC architecture, the source and destination information requirements are defined in EXPRESS and mappings are defined between the two. The architecture is composed of three parts: 1) a front-end of the translator which performs the syntactic translation function. This converts the syntax of the source system's data from its native form into a form defined in EXPRESS. Since the two representations are equivalent, only *syntactic* (homogeneous) translation takes place. 2) A back-end implemented in the same way as the front-end; the code in the back-end performs syntactic translation from the data defined in EXPRESS to the data format expected by the destination system. 3) Mapping code, which specifies the mappings from the data types defined in the source working form to the data types defined in the destination form. Since the source and the destination schemas are inherently different in nature, *semantic* (heterogeneous) translation takes place. Because mapping languages were still under

development by the time the paper was written, the authors described the inter-schema mappings with hand-drawn mapping diagrams and then wrote the code to implement these mappings. They mention the fact that a mapping language such as EXPRESS-M would eliminate the need to do this manually.

Standard Engineering Analysis Representations

The STEP standard currently includes three standards for the exchange and description of engineering analysis information. These are:

- Part 104 (ISO 10303-104 1995): an integrated resource for finite element analysis of linear stress-displacement problems using unstructured meshes;
- Part 105 (ISO 10303-105 1997): an integrated resource for the kinematics analysis of assemblies of rigid bodies with flexible joints;
- AP 209 (Hunten 1997; ISO 10303-209 1996): an application protocol for the exchange of finite element models and results of composite parts. An important feature of AP 209 is the sharing of information between the design and analysis product definitions. Another crucial concept of this AP is that the shape and analysis information is meant to be implemented to enable bi-directional transfer to enable the feedback of information in the iterative design-analysis environment.

These three parts cover an important but small part of the engineering activity. They are limited to stress-displacement analysis using finite elements and kinematics analysis, and do not address other physical phenomena or other analysis methods. In addition, they do not allow for a definition of the analysis problem that is independent of the solution method, nor they support version control for material and environmental properties used in analysis.

In response to this deficiency, ISO is currently coordinating a standardization effort to develop an Engineering Analysis Core Application Resource Model (EA C-ARM) (ISO 1997). This standardization activity will provide a core model to support the common information requirements for many types of engineering analysis. These common information requirements include material data, the modeling of variable properties (including variable shape), and those aspects of configuration control that are important for

the effective management of analysis tasks in the design process. The common information requirements will be expressed as groups of application-specific information entities known in STEP as "Units of Functionality" (UoFs), which can be included within Application Protocols (APs) as required[4]. The EA C-ARM will define information requirements for engineering analysis that are within the scope of existing and future APs. The intent is to provide an interoperating suite of engineering analysis APs that will share information in a multi-disciplinary environment. To date, four APs have been identified for initial implementation within an engineering analysis suite:

- *AP 209:* described above;
- *Material Services AP:* that will provide a standard to represent properties and allowables for materials, adhesives and standard fasteners;
- *Aero-Thermal Elasticity AP:* which will represent information used in the simulation of the interaction between flight vehicle components and the air. This AP will include results generated by Computational Fluid Dynamics (CFD) analysis using finite difference methods on structured grids;
- *Dynamic Mechanisms Analysis AP:* which will represent information used in the dynamic simulation of mechanisms with flexible links;
- *Electro-Mechanical Subsystems AP:* which will represent information to perform the electro-mechanical subsystems integration and analysis tasks such as control laws and state-space analyses.

The scope of the EA C-ARM includes the following analysis information:

- *Material Data:* to include the representation of material information required for engineering analysis, such as material variation with respect to environmental conditions, behavior of material volumes and material surfaces such as creep, fracture, fatigue, and corrosion, material property distribution, composites, allowable values and material fabrication processes.

[4] Each UoF is mapped to the STEP Integrated Resources (Appendix A) by an interpretation process known as "mapping" when it is used in an AP.

- *Property Modeling:* to enable the description of properties that are distributed within a material object. These properties may take the form of material properties, distributed loads or analysis results.
- *Configuration Control and Management Data:* to support expanded version control (to track versions of analysis models, materials, environments, loads and results), problem description and idealization, engineering allowables, and derived properties.

The information entities defined in the EA C-ARM will overlap with the Application Resource Models (ARMs) of existing APs. In these cases, the EA C-ARM will encompass those entities that are generic, providing an opportunity for interoperable extensions in each AP. For example, two current APs that overlap with the EA C-ARM are:

- AP203 (ISO 10303-203 1994): the EA C-ARM will extend the requirements of AP 203 for configuration control to include versioning of loading conditions, material properties and analysis results.
- AP209: the EA C-ARM will include the requirements of AP 209 for the definition of composite layups and for analysis using finite elements. The EA C-ARM will generalize the requirements so that AP 209 will be interoperable with future APs supporting structured analysis meshes and other analysis methodologies.

The EA C-ARM also has close relationships with some of the current generic and integrated resources of the STEP standard. These resources already include some semantics within the context of engineering analysis. For example, Part 42 (ISO 10303-42 1994) already supports the description of the geometry of real or idealized material objects required by the EA C-ARM; the EA C-ARM will extend Part 42 to include parametric volume entities and mathematical representations. Another example is Part 45 (ISO 10303-45 1994), which provides support for describing material properties.

The expected benefits of the EA C-ARM are:

- Reduced development time for engineering APs, since information requirements will not have to be re-invented from scratch for each new AP; and

- Better quality of the information requirements, since they will be reviewed by the domain experts, rather than "lost" within a larger AP.

Another significant standardization effort – previous to STEP's EA C-ARM and with close ties to it – is the Generic Engineering Analysis Model (GEM) Project (ESPRIT 1993; ESPRIT 1996; Helpenstein, Kenny et al. 1997a; Helpenstein, Kenny et al. 1997b). The GEM Project was one of the projects (project 8894, started in June of 1993 and completed in June of 1996) in the context of Computer Integrated Manufacturing and Engineering (CIME) of the European Specific Program for Research and Development in Information Technology (ESPRIT) sponsored by the European Union (EU) Directorate-General XIII. The aim of GEM project was to develop a Generic Engineering Analysis Model which could be used for the exchange, data sharing and archival of engineering analysis models. GEM had to be general enough to support a range of industrial applications, a variety of design and analysis methodologies, and facilitate the use of analysis results in the design model. In order to do this, GEM represents properties and results independent of the analysis method or discretization used, in such a way that they are associated with the underlying geometry or product component. To ensure that GEM was sufficiently generic, a survey of end user industrial problem in the types of engineering analysis which it supported was commissioned. Careful consideration was also given to the need to interface with CAD-generated data. As a result, GEM is capable of supporting the following types of analysis and solution techniques:

- *Analysis types:* structural mechanics, fluid mechanics, thermodynamics and heat transfer, electromagnetic, metallurgical transformations.
- *Solution techniques:* finite element, finite volume, finite difference, boundary element, transmission line, ray tracing.

Since its inception, one of the main tasks of the GEM project was to identify and leverage any engineering analysis capability already existing in the STEP standard, providing extensions to it whenever its capabilities were found inappropriate or insufficient. In order to facilitate this process, GEM used the same methodology as STEP, particularly in its use of EXPRESS and P21 files. The GEM project identified some STEP parts that support some shape and analysis information, but the coverage was found to be very limited. For example,

the only analysis method supported in STEP was finite element. Moreover, it was almost impossible to use STEP for a description of an analysis task and for the representation of numeric functions with arbitrary interpolation rules.

During the lifetime of the GEM project, members of the modeling team participated actively in the development process of STEP. Many decisions in international standardization could in this way be harmonized with the requirements of the GEM. The participation in ISO activities allowed the GEM developers to use up-to-date STEP Parts and consistent methodology, thus guaranteeing that any feedback to standardization was compatible with existing Parts. The GEM Project influenced STEP's engineering analysis standardization work significantly, particularly in the development of the EA C-ARM. In fact, the data model developed for the GEM project served as a starting point for the development of the EA C-ARM, and the members of the GEM team are still actively involved in this ongoing standardization effort. The EA C-ARM enjoys a strong industrial interest, especially from the aircraft industry, which is providing a significant amount of resources to help expedite its development.

Summary of Gaps

In light of the literature survey of the preceding three sections, the following items stand out as needing additional attention:

- *Lack of a product modeling representation tailored to design-analysis integration*

There is a need for a general product representation that addresses the special needs of design-analysis integration and that can be easily defined and modified by an analysis expert and not necessarily by a computer-programming expert. Although some of the works surveyed provide mechanisms for defining modeling objects and the relationships between their attributes, they do not define semantics specific to design-analysis integration. For example, product idealization relations are not clearly distinguished from other types of relations, and concepts that are important to analysis - such as multi-fidelity idealizations (Subsection 5) - are not defined.

In some design-analysis integration approaches analysis applications retrieve the data they need *directly* from design representations. Most design representations, particularly standardized ones such as STEP Application Protocols, are convenient for the *homogeneous* data exchanges that take place between design applications with similar scope and semantics (for example, when sending data from one geometric modeler to another from a different vendor using AP203). However, they are not the most appropriate for the exchange of data between design and analysis applications, because of the large gap in scope and semantics between design and analysis (making it an *heterogeneous* data exchange), and because most of the information contained in the design representations is not needed at all in analysis. As a consequence, the codes of the analysis applications become considerably more complex, since they have to resolve this semantic mismatch themselves and carry out the transformations or idealizations required. Common idealizations that could be potentially shared among several analysis applications are instead replicated in the codes of each individual application. In addition, design representations usually have data structures that are too complex and present information in terms that are unfamiliar to analysis experts.

Some of the approaches surveyed utilize AI and agent-based techniques to idealize the design model and populate the analysis models. These techniques are difficult to generalize, implement and modify and therefore are more appropriate for specific domains and solution methods (for example, finite-element thermomechanical analysis of multi-chip modules). It appears that in only such cases it is feasible to create a knowledge base complete enough to support the idealization process.

- *Lack of a modeling language for defining analyzable views of products*

Associated with the first item, there is a need for a modeling language that allows analysts to easily create, modify or extend analyzable views of products without requiring extensive programming, and that is independent from the domain, solution technique or computer applications used. General-purpose product modeling languages such as EXPRESS do not contain specific semantics to better describe and facilitate design-analysis integration. For example, EXPRESS does not convey concepts such as product idealization, design data sources, design data integration, or multi-fidelity domains.

- *Lack of explicit representation of data integration knowledge*

Also requiring additional attention is the problem of how design information scattered across several repositories is retrieved and properly combined for analysis purposes – a common issue when it comes to design-analysis integration. There is a need for formal, explicit, implementation-independent definitions to describe this data integration process. If standard representations are used as the source of design information, the integration strategy must take into account that there will be invariably some analysis that requires information not supported by any *existing* standard representation. Obviously, it would be impractical to develop a new standard representation for each new analysis situation that arises as the number of these representations would grow exponentially. Therefore, the overall design representation must be considered, in general, as an *aggregation* of standard and non-standard representations.

- *Lack of explicit representation of design idealization knowledge*

There is also a need for a mechanism for formally defining the transformations required to idealize design information. Normally, these transformations are not explicitly defined anywhere and, as a consequence, end up buried inside the code of the analysis applications (or in the minds of the analysts), making it difficult to reuse or modify them.

- *No clear distinction between product and analysis models*

In some of the representations surveyed, analysis models are combined with product models. As a result, there is not a clear distinction between the attributes and relations that belong to the product (in other words, that are *intrinsic* to it) and those that correspond to the analysis model. This distinction is important because intrinsic product attributes and relations are independent of the environmental conditions to which the product is subjected (and therefore portable and reusable), whereas analysis attributes characterize the behavior of the product under specific environmental conditions.

- *Lack of bi-directional idealization transformations*

The idealization approaches surveyed do not explicitly support design synthesis, where the flow of information goes in the reverse direction, that is, from analysis to design. There is a need for a mechanism to define product idealization transformations so that they can also be used for design synthesis (Subsection 5).

- *Limited availability of analysis representation standards*

As mentioned above, current product data exchange standards have not yet provided a general-purpose design-analysis integration representation. As a consequence, most data exchange implementations between design and analysis applications still require significant customization and/or are largely confined to proprietary solutions. The STEP standard currently includes three standards for the exchange and description of engineering analysis information (Part 104, Part 105 and AP 209), but they are limited to specific physical phenomena and analysis methods. Even when the EA C-ARM is completed, applications will not be able to take advantage of it until specific APs that make use of the EA C-ARM are developed, which requires considerable time due to the inherent inertia of the standardization process. Once developed, APs are static representations and changing or extending them requires a long process of discussion and balloting.

CHAPTER 3

PROBLEM STATEMENT AND THESIS OBJECTIVES

"In the middle of difficulty lies opportunity."

(Albert Einstein)

The previous two chapters defined the general problem of design-analysis integration and identified some gaps that exist in the current state of this field. This chapter will provide a statement of the problem that this thesis addressed as an attempt to fill some of these gaps and list the objectives that drove the development of the Analyzable Product Model Representation.

Problem Statement

The problem addressed by this thesis can be stated as follows:

> *There is a need for a formal engineering information representation that addresses the special needs of design-analysis integration. This representation should provide the necessary constructs for defining analysis-oriented views of an engineering part. These analysis-oriented views should provide a single source of analysis information that can be used by a family of diverse analysis models, including multi-fidelity models. The analysis models being supported should drive the semantics and the amount of analysis information that these views are presenting.*
>
> *The representation should also provide a mechanism for explicitly describing and capturing the rules or knowledge used to combine the design information spread across multiple design repositories, as well as the transformations required to idealize this design information.*

Thesis Objectives

The main objective of this work was to develop a formal, generic, computer-interpretable engineering representation that could be used to create analysis-oriented views of engineering parts or products. As illustrated in Figure 27-1, these analysis-oriented views should combine design information spread over multiple design representations and add idealized information, providing a unified perspective of the product that is more suitable for analysis. In addition, this representation should bridge the semantic and syntactic gap between design and analysis representations and enable reusability by supporting data entities and idealizations that can be shared among multiple analyses.

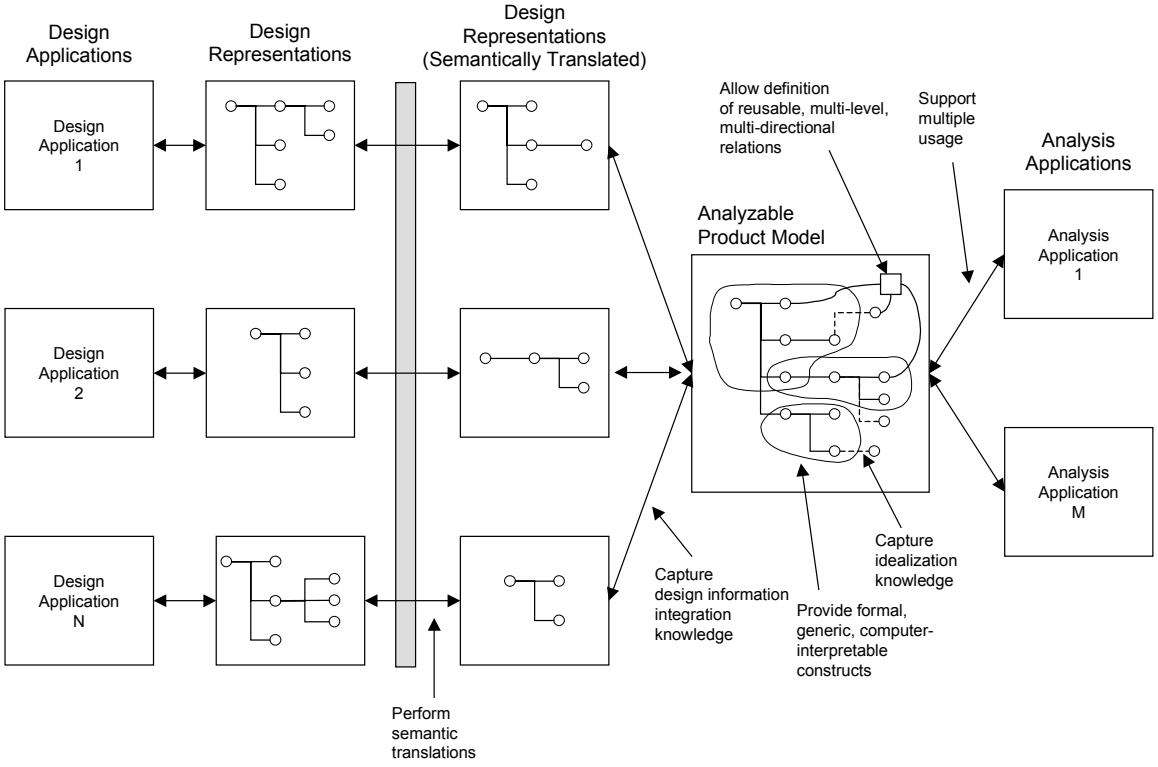


Figure 27-1: Focus of this Thesis: The Analyzable Product Model

This new representation - named Analyzable Product Model (APM) Representation – should provide the building blocks needed for defining analysis-oriented views of products. This

representation should also define the operations that can be used by client applications[5] to access and manipulate the information contained in these views.

What follows is a list of the objectives that drove the development of the APM Representation presented in this thesis. These objectives can also be viewed as the requirements for any product representation whose purpose is to facilitate design-analysis integration. They define the criteria against which the APM representation will be evaluated later in this thesis (Chapter 97). Examples of these needs are given in chapters 1 and 7 and are thus referenced here.

In order to facilitate and enhance the objectivity of the evaluation process, an effort was made to state objectives that are as well defined and independent as possible. Consequently, the list of objectives is rather long. To facilitate understanding, the objectives were grouped into the following eight categories:

1. Analysis-Oriented View Definition Objectives;
2. Multiple Design Sources Support Objectives;
3. Idealization Representation Objectives;
4. Relation Representation and Constraint Solving Objectives;
5. Analysis Support Objectives;
6. Data Access and Client Application Development Objectives;
7. Compatibility with Other Representations Objectives; and
8. General Objectives.

Analysis-Oriented View Definition Objectives

These objectives refer to the ability to define analysis-oriented views of product information to support the unique information requirements of engineering analysis and bridge the semantic gap between design and analysis.

[5] APM client applications are programs that directly access the information defined in an APM. Normally, client applications are *analysis* applications, but they do not *have* to be. An example of a client application that is not analysis application is an application to browse and modify the structure and contents of an APM (it accesses APM data but it is not used to perform any engineering analysis).

Objective 1: *Provide the necessary constructs for defining analysis-oriented views of an engineering part.*

The APM Representation should provide the necessary modeling building blocks for defining analysis-oriented views of product models. These analysis-oriented views should specify how to combine information from several design models and derive idealized information from this design information in order to support the requirements of a set of related analyses[6]. Analysis-oriented views should provide a unified perspective of the product that is more suitable for analysis.

Objective 2: *Bridge the semantic gap between design and analysis representations.*

As discussed in the previous chapter, one of the main problems encountered with current design-analysis approaches is that, in these approaches, analysis applications retrieve the data they need *directly* from design representations. However, design representations are not the most appropriate direct sources of analysis information because of the large gap in scope and semantics that exists between design and analysis. In addition, design representations usually have data structures that are too complex and present information in terms that are unfamiliar to the analysis expert.

The APM representation should provide the capability to define and populate analyzable product models that present analysis information at a semantic level more compatible with the analysis models. This capability should include a mechanism to perform the necessary syntactic and semantic translations to the design data to transform it into analyzable information.

Objective 3: *Enable the creation of concise analyzable product models.*

As discussed in the previous chapter, much of the information contained in design representations is not needed at all in analysis. One of the driving reasons for resorting to an analyzable product model is to simplify the design information and make it more compatible with the needs of the analysis models that are going to use it. Hence, analyzable product models defined with this representation should contain only the information needed by a

[6] In this context, the term "related analyses" refers to analyses that evaluate similar or interrelated phenomena of the same product. For example, a set of related thermomechanical analyses for PWBs.

family of related analyses. In most cases, this may involve eliminating or simplifying design information. One example of information that is often simplified is geometry, since the detailed geometric representation created during design is rarely needed in analysis.

Objective 4: *Allow easy creation, modification and extension of analyzable product models.*

Analyzable product models should be easy to create, modify and extend. The syntax used for creating these models should be easy to understand by the people who are familiar with the knowledge being represented. Ideally, the same experts familiar with the domain being represented should be able to develop an analyzable product model with little assistance from a data-modeling expert.

Multiple Design Sources Support Objectives

These objectives refer to the ability to combine design data coming from different sources in order to support engineering analysis, and to explicitly represent the rules used to combine this data into a single integrated representation.

Objective 5: *Support for multiple sources of design information.*

One of the main premises of this thesis is that the information required for analysis spans multiple design repositories. For this reason, the APM representation should provide some method to specify how design information should be retrieved from multiple sources and combined to create a unified, analyzable view of it.

Objective 6: *Allow explicit representation of design data integration knowledge.*

One of the main gaps in current design-analysis integration approaches identified in the previous chapter is that the rules used to combine the various design sources are not explicitly captured anywhere. As a result, they end up buried inside the codes of the analysis applications, becoming very difficult to maintain and reuse.

The APM Representation should provide some mechanism to capture these integration rules as part of the analyzable product model itself. By doing so, it will not be necessary to replicate code to implement these rules in each analysis application. The APM

Representation should provide formal, implementation-independent definitions to explicitly specify how design data from multiple sources is combined. These definitions should be easy to recognize, modify and reuse.

Idealization Representation Objectives

These objectives refer to the ability to describe idealized information about an engineering part in order to be used by analysis models and to explicitly represent the transformations required to obtain this idealized information from design information.

Objective 7: *Allow explicit representation of idealization knowledge.*

In current typical approaches, as discussed in the previous chapter, the transformations required to obtain the values of idealized attributes from design or product attributes are not explicitly defined anywhere. As a consequence, they end up buried inside the code of the analysis applications making it difficult to reuse or modify them. The APM Representation should provide the necessary constructs for defining idealized attributes or features of the part as well as the mathematical relations that define how these idealized attributes are derived from the "real" or "manufacturable" attributes of the part. These definitions should be formally captured *as part of the analyzable product model itself*.

Objective 8: *Allow the definition of reusable idealizations.*

Idealized attributes and product idealization transformations should be defined in such a way that they can be used by potentially more than one analysis application (in other words, be *reusable*). This is illustrated in Figure 1-4, where a linkage has been idealized as an I-section truss. This idealized view of the linkage is being used by two applications: a tension analysis application and a torsion analysis application. The ability of reusing idealizations will avoid having to replicate idealization code in each analysis application.

Objective 9: *Allow the definition of multi-fidelity idealizations.*

Often, analysts perform the same analysis at different levels of precision by using more or less accurate idealizations of a feature. For instance, a coarse analysis may only require a

simple approximation of a feature, whereas a more precise analysis would require a more detailed (and, consequently, more computationally-demanding) version.

Hence, the APM Representation should allow the definition of multiple fidelity levels of the same idealized feature. For example, as shown in Figure 1-4, the same linkage of the previous example has been idealized as a straight I-section truss (as in the previous example) and also as a half I-beam of variable height with to half sleeves at each end. Both idealized views of the linkage should be available to any analysis application accessing the analyzable product model.

Relation Representation and Constraint Solving Objectives

These objectives refer to the ability to define mathematical relations (or constraints) among the attributes of an engineering part. It also refers to the ability to use these relations to solve for the unknown value of one or more attributes.

Objective 10: *Allow the definition of complex relations.*

Relations are mathematical constraints that relate the values of the attributes of a part or feature. The APM Representation should provide the capability to define systems of relations of relative complexity. In addition to the common algebraic operations (addition, subtraction, multiplication, division), it should be possible to define relations that involve transcendental functions (trigonometric, exponential, logarithmic, etc.), powers, absolute values, aggregate operations (sums, averages, minimums, maximums), conditional (`if-then`) statements, counter-controlled repetitions (`for` or `while` loops) and calls to external procedures.

Objective 11: *Allow the definition of multidirectional relations.*

A true design-analysis integration environment requires a bi-directional integration between design and analysis. In other words, the flow of information between design and analysis representations via an analyzable product model should not be limited to one direction (for example, *from* design *to* analysis). In the case of design checking, in which analysis is used to check a particular design, the flow of information is from design to analysis. However, when

analysis results are used to define the design (as is the case of design synthesis) the flow is in the opposite direction. The APM Representation should support both directions.

In order to support this bi-directional flow of information between design and analysis, the APM Representation should provide the capability of dynamically changing the input/output directions of the relations defined in an analyzable product model. For this purpose, the definition of a given relation should not dictate which of the attributes participating in the relation are inputs or outputs.

Objective 12: *Allow dynamic relaxation of relations.*

In some special cases, the constraints imposed by a given relation may make reaching a satisfactory design unnecessarily difficult or even impossible. In these cases, the analyst may want to consider ignoring (or "relaxing" or "weighting") the obstructing relation in order to provide an additional degree of freedom in a particular design. Hence, the APM Representation should provide the capability to dynamically relax or temporarily remove a relation from the APM without having to permanently delete.

Objective 13: *Support for multiple constraint solvers.*

Specific implementations of the APM Representation (see **Objective 18**) will require the services of a constraint-solving system to solve the relations defined in an analyzable product model. It should be possible to use any constraint solving system (internal or external) in conjunction with a specific implementation of the APM Representation. In other words, implementations of the APM Representation should not be tied to any particular constraint solving system. Moreover, it should be easy to replace one constraint solver with another within the same implementation.

Objective 14: *Allow constraint solver-independent definition of relations.*

The syntax used to define relations in an analyzable product model should be independent from the syntax used by the specific constraint solver system being used (see **Objective 13**). It should be possible to map the syntax used to define relations in an analyzable product model into the syntax of any specific constraint-solving system.

Objective 15: *Allow easy definition and modification of relations.*

The APM Representation should provide an easy way to add relations to an analyzable product model or to modify existing ones. The codes of the applications using the analyzable product model should not need to be modified to reflect change or additions of relations.

Analysis Support Objectives

These objectives refer to the ability to support the information requirements of diverse engineering analysis models and solution methods.

Objective 16: *Allow support for multiple analysis models and solution methods.*

The APM Representation should provide a mechanism for defining a single source of information that supports the need of a *set* of related analyses (**Objective 1**). In general, the analyses in these sets will be based on different analysis models. In addition to multiple analysis models, these analyses may also use different solution methods to reach a solution (e.g., formula-based, finite-element, etc.). The choice of a particular combination of analysis model and solution method will depend on the level of accuracy desired and the computer resources available. The information requirements of these different analysis models and solution methods may vary. The APM Representation should allow for the creation of APMs that support the information requirements of multiple analysis models and solution methods.

Objective 17: *Provide flexibility to easily add additional analyses.*

It should be easy to add a new analysis to the suite of analyses currently being supported by an analyzable product model. A new analysis may place additional information requirements on the analyzable product model by adding any of the following:

1. New phenomena (or combination of phenomena) being investigated.
2. New analysis models.
3. New solution methods.

Data Access and Client Application Development Objectives

These objectives refer to the ability to develop analysis applications that access APM-defined information through a set of well-defined operations and to take advantage of the APM Representation to simplify the codes and enhance the maintainability of these analysis applications.

Objective 18: *Provide a set of operations to access APM-defined information.*

The APM Representation should provide a reasonable set of data-access operations that can be performed on an analyzable product model. Implementations of these operations in specific programming languages will result in some form of library of software components that can be used to develop client applications. Collectively, these data-access operations will provide a protocol through which client applications can access information about the structure of an analyzable product model as well as particular instances of data conforming to this structure.

The operations provided should support critical design-analysis integration tasks such as loading the definition of the analyzable product model, loading and combining the design data, using the design data for analysis, and saving changes.

Objective 19: *Allow the definition of late-bound operations.*

Late-bound operations are designed to manipulate APM information without previous knowledge of the domain-specific structure of the data. It should be possible to reuse these operations in a range of application domains without having to modify or customize them. More importantly, they should allow the development of *APM-generic applications*: applications designed to work with *any* domain-specific APM. Examples of potential APM Generic Applications that could be developed using late-bound operations are APM Browsers, APM Integrated Development Environments, and APM Conformance-Checking Tools.

Objective 20: *Reduce the complexity of analysis code.*

When using the services of an analyzable product model, analysis applications should not have to include code to combine the information coming from multiple design sources nor should they have to include code to carry out the transformations to idealize design information. In addition, the semantic mismatch between the design representations and the analysis representation on which the application is based should already be resolved in the analyzable product model (**Objective 2**). As a result, the codes of the APM-based analysis applications should be considerably simpler than the codes of those that do not.

Objective 21: *Isolate analysis applications from the format of the design data.*

The APM Representation should provide some mechanism to isolate the code of the analysis applications from the choice of data format in which the design information is stored. In other words, the choice of data formats should not affect the code of the analysis applications. In addition, it should be possible to switch from one data format to another without having to modify the codes of the analysis applications.

Objective 22: *Allow development of constraint solver-independent client applications.*

The APM Representation should provide a mechanism to isolate the code of the analysis applications from the constraint solver being used. In other words, the choice of constraint solver used in a specific implementation should not affect the code of the analysis applications. In addition, it should be possible to switch from one constraint solver to another without having to modify the code of the analysis applications.

Objective 23: *Hide constraint-solving details from client applications.*

The operations specified by the APM Representation (see **Objective 18**) should handle constraint-solving details such as:

1. Deciding *when* to solve for an unknown or idealized value;
2. Deciding *which relations* to use;
3. Prepare the constraint-solving request for the specific solver;

4. Running the constraint solving process; and
5. Receiving and interpreting the results returned by the constraint solver.

These actions should be completely transparent to the developers of the analysis applications. In other words, developers of analysis applications using the APM Representation should not have to write any code to handle the constraint-solving details listed above.

Compatibility Objectives

These objectives refer to the ability to exchange information between the APM Representation and other representations (standard or otherwise).

Objective 24: *Leverage existing product data exchange standards.*

The APM Representation should leverage the ability of existing standard product data representations such as STEP (ISO 10303) to represent design information in a neutral way. The APM Representation should be able to read design information conforming to these standards.

Objective 25: *Support multiple design data formats.*

The analyzable product model should be able to combine design information stored in multiple formats. If standard representations (such as STEP or IGES) are used as the source of design information, the integration strategy used by the APM Representation must take into account that there will be invariably some analysis that requires information not supported by any *existing* standard representation. Obviously, it would be impractical to develop a new standard representation for each new analysis situation that arises, as the number of these representations would grow exponentially. Therefore, the overall design representation must be considered, in general, as an *aggregation* of standard and non-standard (proprietary or native) representations.

Objective 26: *Compatible with existing CAD/CAE Tools.*

From the point of view of the APM, CAD tools *create* the design data that populates the APM whereas CAE tools *consume* this design data and the idealized information added in the APM. Hence, compatibility with existing CAD tools means that the APM Representation should be able to read the design data created by these tools. On the other hand, compatibility with existing CAE tools means that the populated APM should be usable with these tools.

Objective 27: *Compatible with the Multi-Representation Architecture (MRA).*

The APM Representation should be compatible with the MRA approach developed at the Georgia Institute of Technology by Drs. Russell S. Peak and Robert E. Fulton (see Chapter 7). More specifically, the APM Representation should be able to complement the MRA by providing the product information required by PBAMs, thus filling the gap between design tools and PBAMs. The APM Representation should provide a mechanism to allow PBAMs to access the information contained in an APM.

General Objectives

This group contains general objectives for the APM Representation.

Objective 28: *Be product domain-independent.*

The APM Representation should be independent from any particular product domain or industry (for example, airplane structures, printed wiring assemblies, etc.). In other words, it should be *generic*. The constructs defined in this representation should not be expressed in terms of any particular domain. The APM Representation should serve as a "template" for creating domain-specific analyzable product models.

Objective 29: *Provide unambiguous and formal definitions.*

The APM Representation should provide unambiguous and formal definitions of the different building blocks used to create and use analyzable product models. These definitions should be independent from any particular data modeling or programming language. They

should state the meaning of these constructs, the information they contain and their engineering significance. The APM Representation must have a well-defined structure composed of a pre-defined vocabulary of symbols. The logical pieces of the representation should be defined as well as rules on how those pieces can be assembled together.

Objective 30: *Have a computer-interpretable form.*

The APM Representation must have some type of computer-interpretable language for defining analyzable product models. A computer program should be able to parse this definition and create a corresponding representation of the analyzable product model in memory that can be accessed and manipulated by analysis applications. This computer-interpretable language must be easy to understand by humans without extensive knowledge of its syntax.

Objective 31: *Have some type(s) of graphical form(s).*

The APM Representation should provide a graphical form (or a combination of graphical forms) that can be used as visual tools for developing, communicating and documenting analyzable product models. The nomenclature used in these graphical forms must be simple and intuitive.

Objective 32: *Provide correct results.*

The APM Representation would not be of any practical values if it did not produce correct results. The values obtained for any derived or idealized attribute should be consistent with the relations defined in the analyzable product model. Note that this objective does not refer to the correctness of the analysis results, but only to the calculation of derived or idealized attributes within an analyzable product model. Of course, if the value of an idealized attribute (for example) is calculated incorrectly, the result of the analyses that use this value will also be incorrect.

CHAPTER 4

THE ANALYZABLE PRODUCT MODEL REPRESENTATION

This chapter provides a formal presentation of the Analyzable Product Model Representation and its different components. The chapter begins describing a design-analysis integration example using the APM Representation (Section 39), with the purpose of providing an overview of the APM approach and the concepts introduced later in the chapter. Next, Section 40 overviews the four main components of the APM Representation (APM Information Model, APM Definition Languages, APM Graphical Representations and APM Protocol) which are formally presented in the remaining sections of the chapter. Section 41 introduces the APM Information Model, which contains the fundamental building blocks of the APM Representation. Section 51 introduces the two definition languages developed in this work - APM-S and APM-I - used to define APMs and APM instances, respectively. Section 54 introduces three graphical representations used to represent APM concepts (APM EXPRESS-G Diagrams, APM Constraint Schematics Diagrams, and APM Constraint Network Diagrams). Section 58 describes a group of APM information-access operations collectively known as the APM Protocol.

This chapter is a self-contained, implementation-independent presentation of the conceptual aspects of the APM Representation, containing the main theoretical contribution of this thesis. A prototype implementation of the concepts introduced in this chapter is presented in Chapter 64, and several test cases validating these concepts – using the prototype implementation of Chapter 64 – are presented in Chapter 83. Figures 38-13 and 38-14 – introduced in Section 40 – provide a roadmap for the concepts presented in this and the next chapters.

Design-Analysis Integration Using the APM Representation

This section overviews how the APM Representation fits in a design-analysis integration scenario such as the one discussed in Section 2. The purpose of this overview is to provide a general context for the sections that follow, which describe the individual components of the APM Representation in detail.

One of the earlier test cases developed by the author will be used as an example to help introduce and demonstrate some of the basic APM concepts. This test case demonstrates how the APM Representation is used in the design and analysis of a hypothetical linkage used in the mechanism of an airplane wing flap ("flap link", for short) such as the one shown in Figure 38-1.

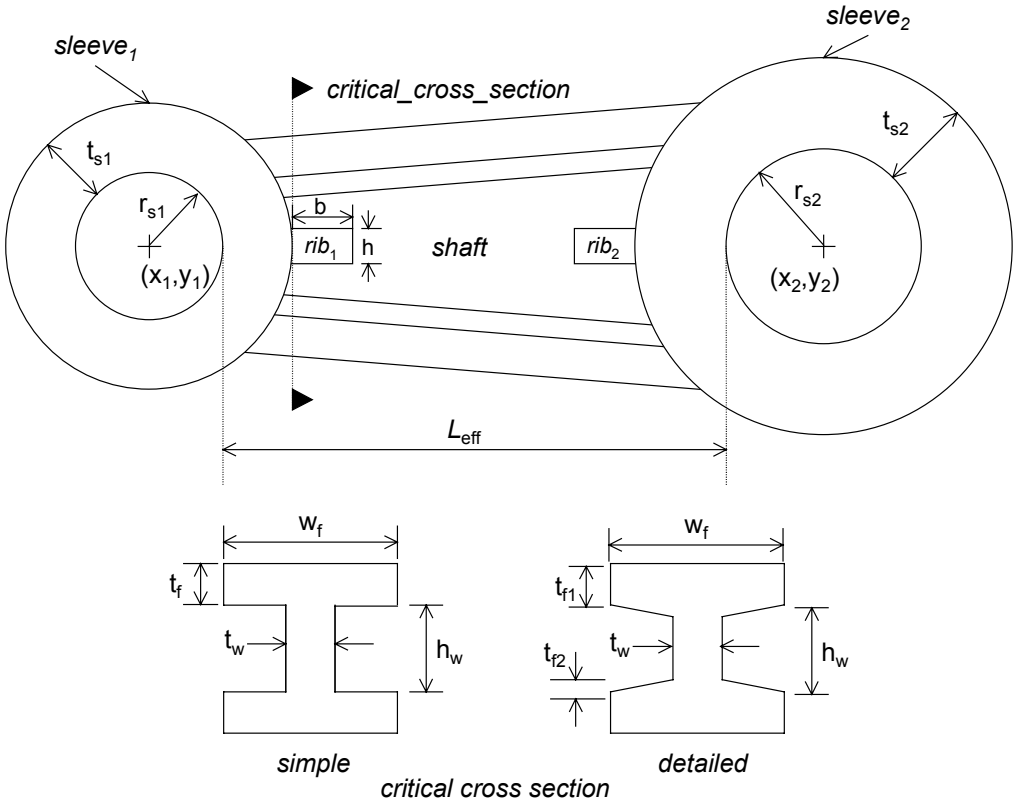


Figure 38-1: Airplane Wing Flap Linkage ("Flap Link")

The design-analysis scenario of this example is illustrated in Figure 38-2 and involves two design applications (a solid modeler and a materials database manager). Each design application creates information about a different aspect of the product: the solid modeler creates geometric information and the materials database manager creates a database of the detailed properties of materials available for the fabrication of the flap link. This information is stored in two separate design repositories (labeled "Geometric Data" and "Material Data").

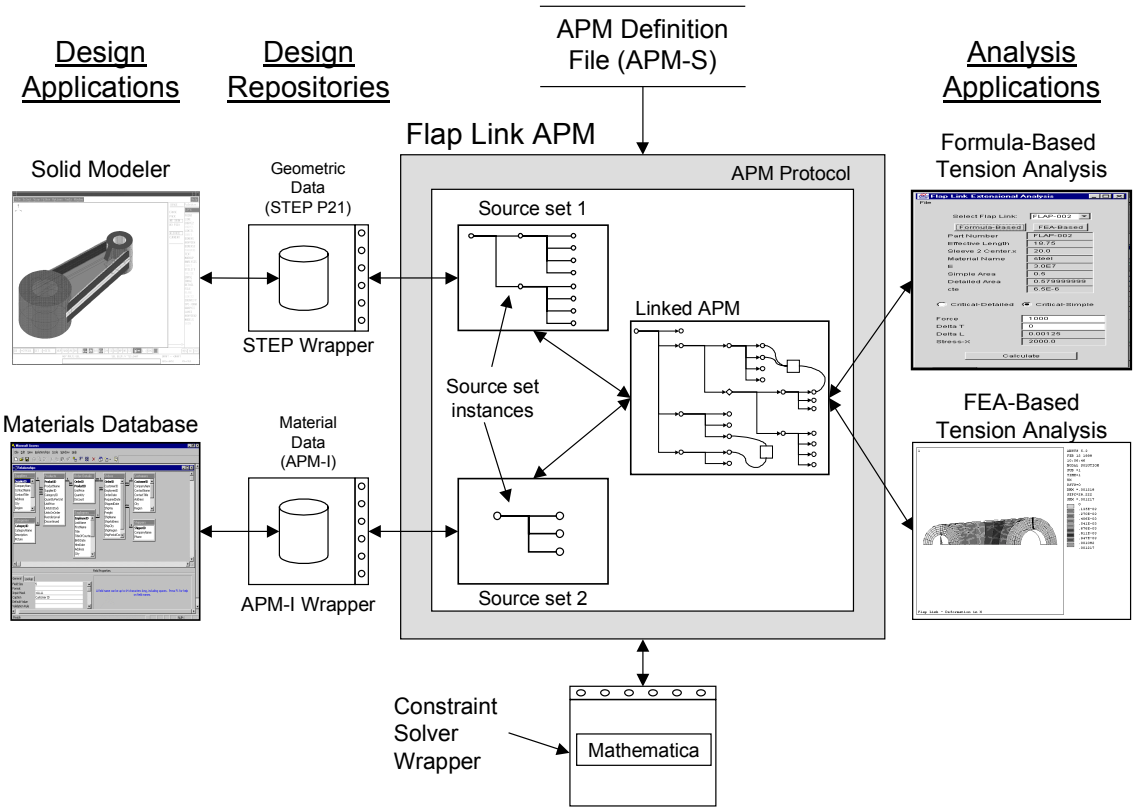


Figure 38-2: Flap Link Design-Analysis Integration Using the APM

In general, design information created by design applications may be stored in a variety of data structures and file formats (both standard and native). For example, in this test case the

geometric information is stored in a STEP Part 21 file[7] (Figure 38-3), whereas the material properties are stored in an APM-I file[8] (Figure 38-4).

```
DATA;
#10=FLAP_LINK( 'FLAP-001' , * , #20 , #40 , #60 , #100 , #110 , 'aluminum' );
#20=SLEEVE( 1.5 , 0.5 , 0.5 , #30 );
#30=COORDINATES( 0.0 , 0.0 );
#40=SLEEVE( 2.0 , 0.6 , 0.75 , #50 );
#50=COORDINATES( 15.5 , 0.0 );
#60=BEAM( #70 , * );
#70=CROSS_SECTION( #80 , #90 );
#80=DETAILED_I_SECTION( * , 0.1 , 0.1 , * , * , * , 0.15 );
#90=SIMPLE_I_SECTION( * , * , * , * , * );
#100=RIB( 10.0 , 0.5 , * );
#110=RIB( 10.0 , 0.5 , * );

#120=FLAP_LINK( 'FLAP-002' , * , #130 , #150 , #170 , #210 , #220 , 'steel' );
#130=SLEEVE( 1.5 , 0.5 , 0.5 , #140 );
#140=COORDINATES( 0.0 , 0.0 );
#150=SLEEVE( 2.0 , 0.6 , 0.75 , #160 );
#160=COORDINATES( 20.00 , 0.0 );
#170=BEAM( #180 , * );
#180=CROSS_SECTION( #190 , #200 );
#190=DETAILED_I_SECTION( * , 0.1 , 0.1 , * , * , * , 0.15 );
#200=SIMPLE_I_SECTION( * , * , * , * , * );
#210=RIB( 10.0 , 0.5 , * );
#220=RIB( 10.0 , 0.5 , * );
```

(* = Unknown value)

Figure 38-3: Flap Link Geometric Data File (STEP P21)

```
DATA;

INSTANCE_OF material;
  name : "steel";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 30000000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.30;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.0000065;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 275.00;
END_INSTANCE;

INSTANCE_OF material;
  name : "aluminum";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 10400000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.000013;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 156.00;
END_INSTANCE;

INSTANCE_OF material;
  name : "cast iron";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 18000000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.000006;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 125.00;
END_INSTANCE;

END_DATA;
```

Figure 38-4: Flap Link Material Data File (APM-I)

[7] STEP Part 21 is the physical data exchange format of the STEP standard (Appendix A).

[8] The APM-I format was developed for this work and is introduced in Section 53.

The design information is used, as shown on the right side of Figure 38-2, to drive two analysis applications. Both analysis applications (Figures 38-5 and 38-6) are used to estimate the change in length and the axial stress of the flap link due to an applied extensional force. The two analyses differ in their *solution methods* and *degree of fidelity*: one is 1D formula-based and the other is 2D finite-element based.

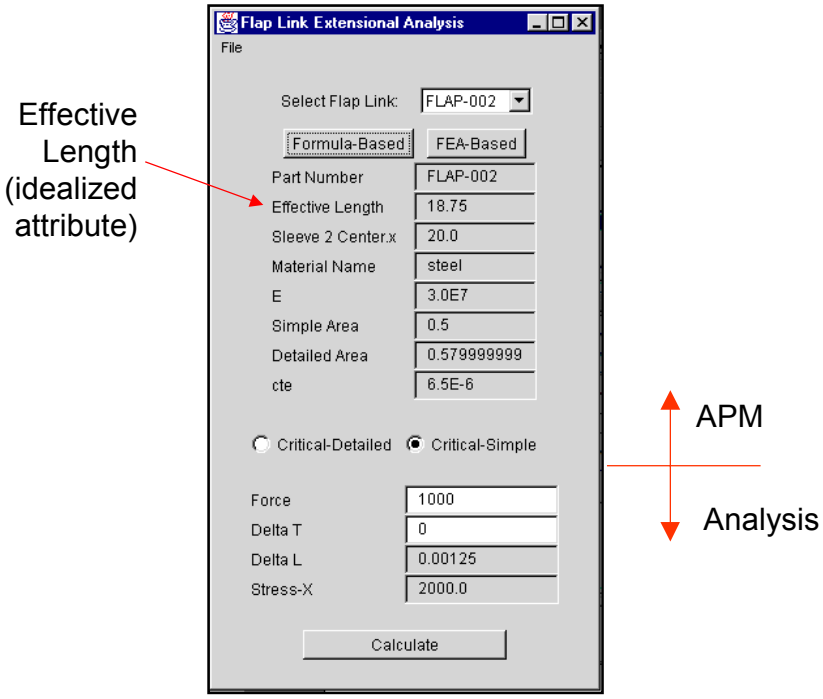
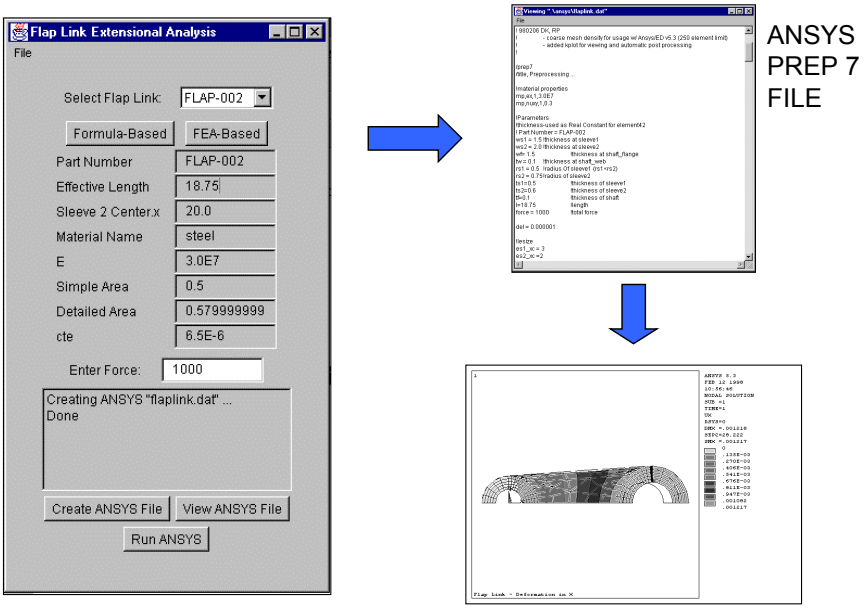


Figure 38-5: Formula-Based Flap Link Tension Analysis



Note: The window does not display all the attributes used in the FEA.

Figure 38-6: FEM-Based Flap Link Tension Analysis

As shown in Figure 38-2, an APM ("Flap Link APM") is located between the design and the analysis applications, providing a single, integrated source of analysis-oriented product information. *Both* analysis applications and *both* design applications read and write information from and to this single source.

This APM is defined using a special modeling language - developed for this work and presented in Subsection 52 - called the APM Structure Definition Language (APM-S). With this language, developers define the source sets, domains, attributes, relations and source set links that make up the structural definition of the APM. The APM Definition is stored in the APM Definition File shown on the top portion of Figure 38-2 and in detail in Figures 38-7 and 38-8. A graphical view of the APM Definition, using the Constraint Schematics Representation is shown in Figure 38-9[9]. This representation shows the different domains defined in the APM (such as `flap_link`, `sleeve`, `beam`, etc.), their attributes (such as `effective_length`, `sleeve_1`, `material`, etc.) and some of the design and idealization relations among them ("`pir1`", "`pir2`", "`pir12`" and "`pr2`").

[9] The constraint schematics diagrams used in this figure will be formally introduced in Subsection 56. Meanwhile, refer to Appendix I for a brief summary of the basic nomenclature.

```
APM flap_link;

SOURCE_SET flap_link_geometric_model ROOT_DOMAIN flap_link;

DOMAIN flap_link;
  ESSENTIAL part_number : STRING;
  IDEALIZED effective_length : REAL;
  sleeve_1 : sleeve;
  sleeve_2 : sleeve;
  shaft : beam;
  rib_1 : rib;
  rib_2 : rib;
  ESSENTIAL material : STRING;
 PRODUCT_RELATIONS
  pr1 : "<rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2";
  pr2 : "<rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2";
 PRODUCT_IDEALIZATION_RELATIONS
  pir1 : "<effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> - <sleeve_1.radius> -
          <sleeve_2.radius>";
  pir2 : "<shaft.wf> == <sleeve_1.width>";
  pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> - <shaft.tf> )";
  pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness> - <sleeve_2.thickness>";
END_DOMAIN;

DOMAIN sleeve;
  ESSENTIAL width : REAL;
  ESSENTIAL thickness : REAL;
  ESSENTIAL radius : REAL;
  center : coordinates;
END_DOMAIN;

DOMAIN coordinates;
  ESSENTIAL x : REAL;
  ESSENTIAL y : REAL;
END_DOMAIN;

DOMAIN beam;
  critical_cross_section : MULTI_LEVEL cross_section;
  length : REAL;
  ESSENTIAL tf : REAL;
  ESSENTIAL tw : REAL;
  ESSENTIAL t2f : REAL;
  ESSENTIAL wf : REAL;
  ESSENTIAL hw : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir5 : "<critical_cross_section.detailed.tf> == <tf>";
  pir6 : "<critical_cross_section.detailed.tw> == <tw>";
  pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
  pir8 : "<critical_cross_section.detailed.wf> == <wf>";
  pir9 : "<critical_cross_section.detailed.hw> == <hw>";
END_DOMAIN;

MULTI_LEVEL_DOMAIN cross_section;
  detailed : detailed_I_section;
  simple : simple_I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir10 : "<detailed.wf> == <simple.wf>";
  pir11 : "<detailed.hw> == <simple.hw>";
  pir12 : "<detailed.tf> == <simple.tf>";
  pir13 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;

DOMAIN simple_I_section SUBTYPE_OF I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir14: "<area> == 2*<wf>*<tf> + <tw>*<hw>";
END_DOMAIN;

DOMAIN detailed_I_section SUBTYPE_OF I_section;
  IDEALIZED t1f : REAL;
  IDEALIZED t2f : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir15: "<area> == <wf>*( <t1f> + <t2f> ) + <tw>*( <t2f> - <t1f> ) +
          <tw>*<hw>";
  pir16: "<t1f> == <tf>";
END_DOMAIN;

DOMAIN I_section;
  IDEALIZED wf : REAL;
  IDEALIZED tf : REAL;
  IDEALIZED tw : REAL;
  IDEALIZED hw : REAL;
  IDEALIZED area : REAL;
END_DOMAIN;

DOMAIN rib;
  ESSENTIAL base : REAL;
  ESSENTIAL height : REAL;
  length : REAL;
END_DOMAIN;

END_SOURCE_SET;
```

Figure 38-7: Flap Link Test Case APM Definition File

```
SOURCE_SET flap_link_material_properties ROOT_DOMAIN material;

DOMAIN material;
  ESSENTIAL name : STRING;
  stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN material_levels;
  temperature_independent_linear_elastic : linear_elastic_model;
  temperature_dependent_linear_elastic : temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;

DOMAIN linear_elastic_model;
  IDEALIZED youngs_modulus : REAL;
  IDEALIZED poissons_ratio : REAL;
  IDEALIZED cte : REAL;
END_DOMAIN;

DOMAIN temperature_dependent_linear_elastic_model;
  IDEALIZED transition_temperature : REAL;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
  flap_link_geometric_model.flap_link.material == flap_link_material_properties.material.name;
END_LINK_DEFINITIONS;

END_APM;
```

Figure 38-8: Flap Link Test Case APM Definition File (continued)

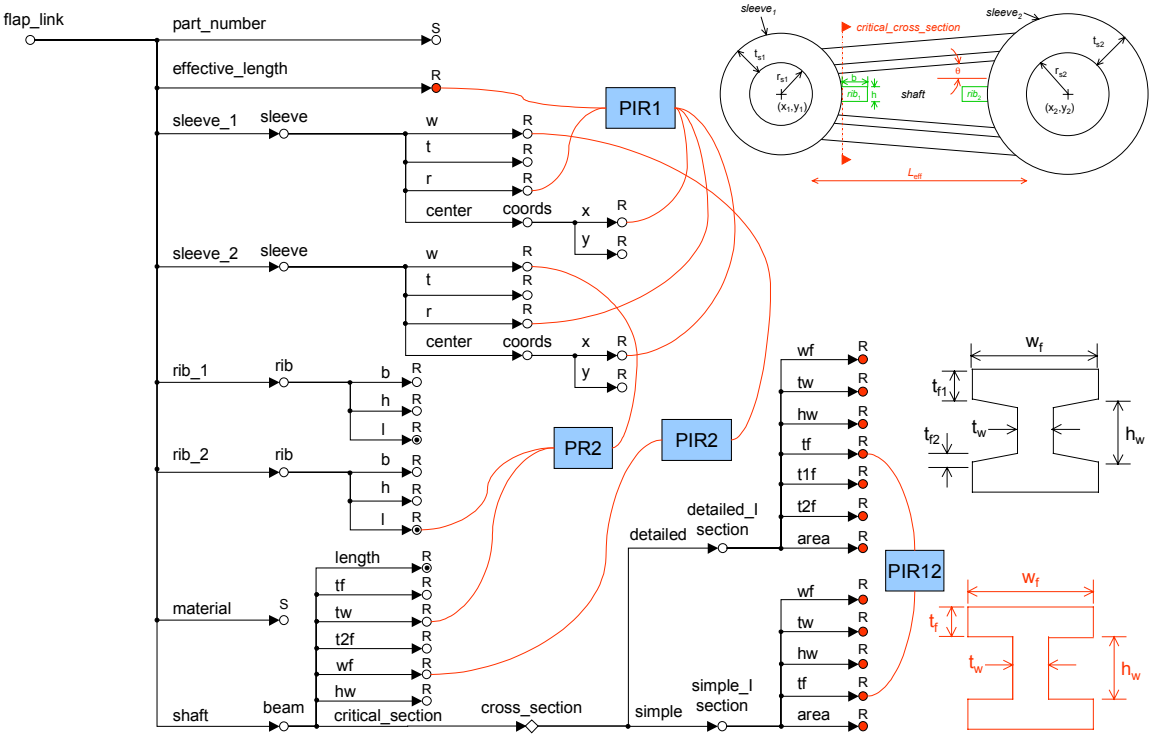


Figure 38-9: Flap Link Test Case APM Constraint Schematics (only `flap_link_geometric_model` source set shown)

An important characteristic of APMs is that they only define attributes (idealized or not) that are *intrinsic* to the part or product as discussed later in Subsection 47.

Continuing with the example of Figure 38-2, the information from the individual design repositories is loaded into the APM. In order to do this, the APM loading operations use the services of special objects called Source Data Wrappers, which "know" the formatting details of the design data. In this test case, for example, there is a Source Data Wrapper to read STEP data (STEP Wrapper) and another to read APM-I data (APM-I Wrapper). These wrapping objects read the design data, perform the necessary conversions, and pass it to the APM in a neutral form understood by the APM. This source data wrapping technique is introduced in Subsection 60 and discussed in more detail in Subsection 79.

As the data stored in the design repositories is loaded into the APM, corresponding APM instances are created. These instances, labeled "Source Set Instances" in the figure, are still grouped in the APM by their source of origin (that is, instances coming from the same design repository are grouped together in the same source set). The next step is to *link* these

instances, according to linking rules defined in the APM Definition. For example, in this case the material name of the flap link is replaced with a material object when the names of both are the same. The result of this linking operation is a single, unified set of instances (labeled "Linked APM Instances" in the figure). The APM Data Linking Operation is introduced in Subsection 60 and discussed in detail in Subsection 79.

Once the design instances are loaded and linked, analysis applications may access them through a specific set of access functions collectively known as in the APM Protocol. Values for derived or idealized attributes (attributes not created by the design applications but needed for analysis) are computed in the APM as they are requested by the analysis applications. For example, the formula-based analysis application of the flap link example requires the value of an idealized attribute called "effective length" (Figure 38-5). Since effective length is an idealized attribute of the flap link, its value is not populated by the design tool (that is, it does not have a value in the design model of Figure 38-3). However, the APM Definition File specifies the mathematical relation needed to calculate its value given the coordinates of the centers of the two sleeves of the flap link (product idealization relation "pir1", in domain "flap_link", Figure 38-9). Thus, when the analysis application executes the following operations from the APM Protocol querying the value of the effective length (**`L_eff`**):

```
L_eff = flapLinkInstance.getRealInstance( "effective_length"
   ).getRealValue()
```

the APM sends, behind the scenes, the relations and the values needed to calculate the effective length to an external constraint solver (Wolfram Research's Mathematica (Wolfram 1996) in this example). The constraint solver solves the system of equations and returns the value of the effective length back to the APM. A wrapping approach similar to the one used to read design data, is also used for constraint solving: an object called **`APMSolverWrapper`** wraps the constraint solver and handles the communication between the solver and the APM. The **`APMSolverWrapper`** receives a request from the APM to solve a system of equations, translates these requests into the appropriate solver-specific commands, runs the solver, gets the results, and sends them back to the APM in a neutral form specified in advance. The Constraint Solver Wrapping Technique used in the APM is discussed in detail in Subsection 81.

Some analysis applications will require more APM information than others, depending on their degree of fidelity and the analysis models on which they are based. For example, Figures 38-10 and 38-11 show the APM information required by the formula- and FEA-based flap link tension analyses, respectively. As expected, the FEA-based analysis requires more detailed information about the flap link than the simpler, less accurate formula-based analysis.

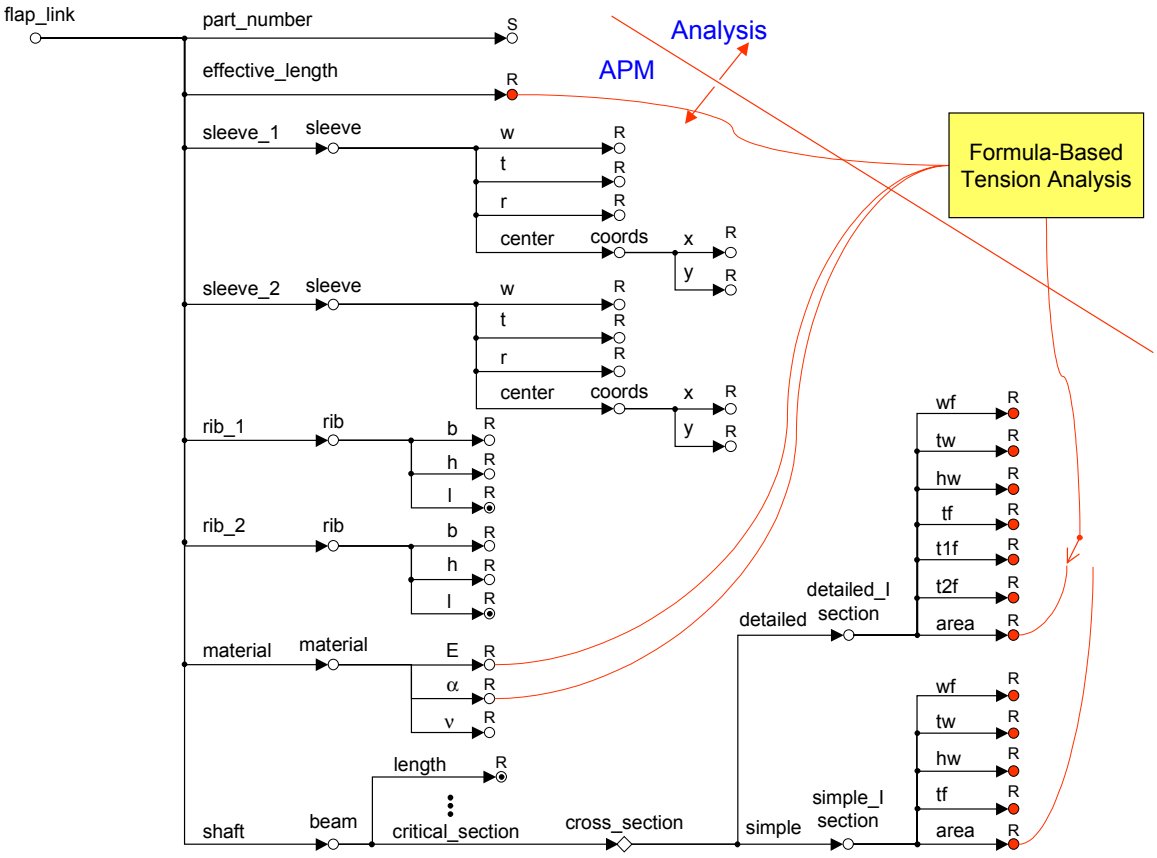


Figure 38-10: APM Information used by the Formula-Based Flap Link Tension Analysis

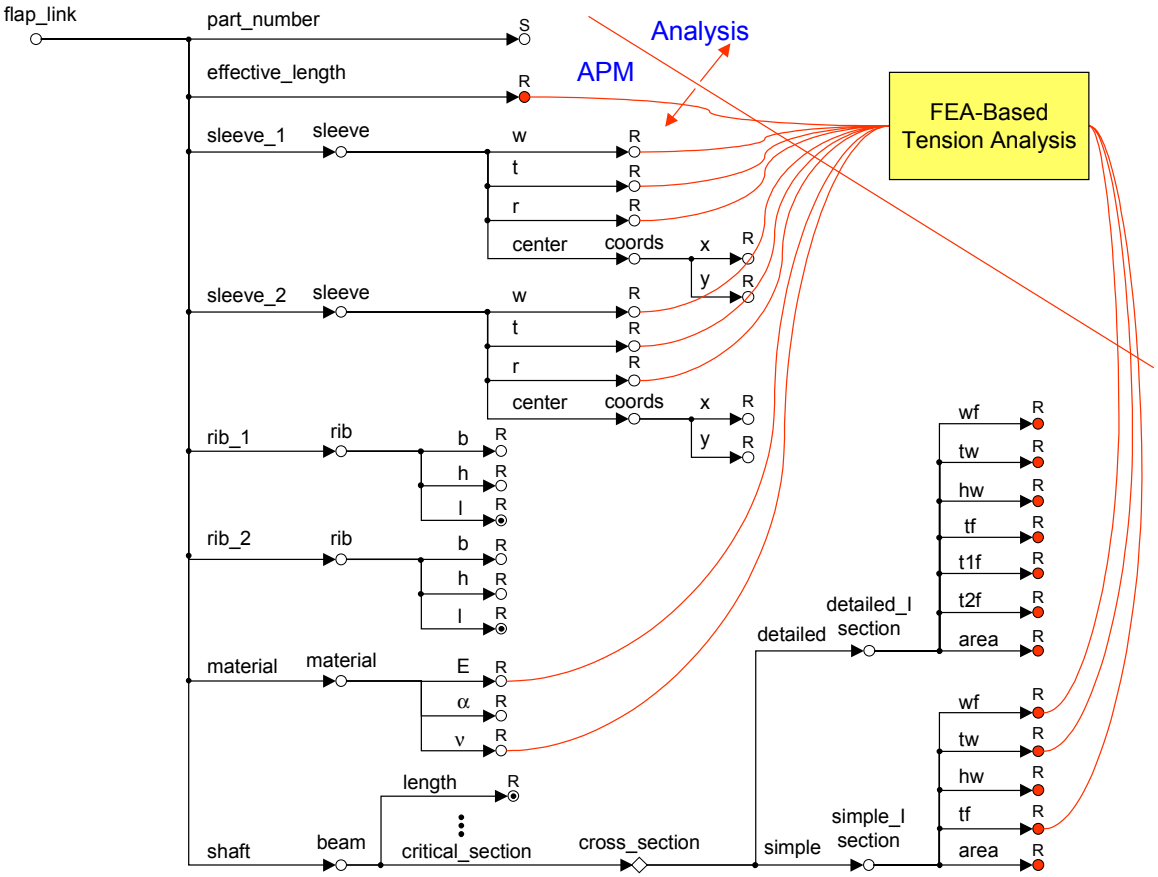


Figure 38-11: APM Information used by the FEA-Based Flap Link Tension Analysis

This example also illustrates two important features of the APM regarding analysis idealizations. The first is the APM's support for *multi-fidelity idealizations*. As shown in Figure 38-12, the formula-based analysis of this test case uses an idealized attribute called "critical cross section" belonging to the shaft of the flap link. As the figure shows, there are two choices for this critical cross section: an approximate or "simple" version in which the critical cross section is simplified as a straight I-Beam, and a more detailed version that takes the variable thickness of the flanges into account. The detailed version represents the actual design, and the simple one illustrates how these can be one or more idealized views of this design feature.

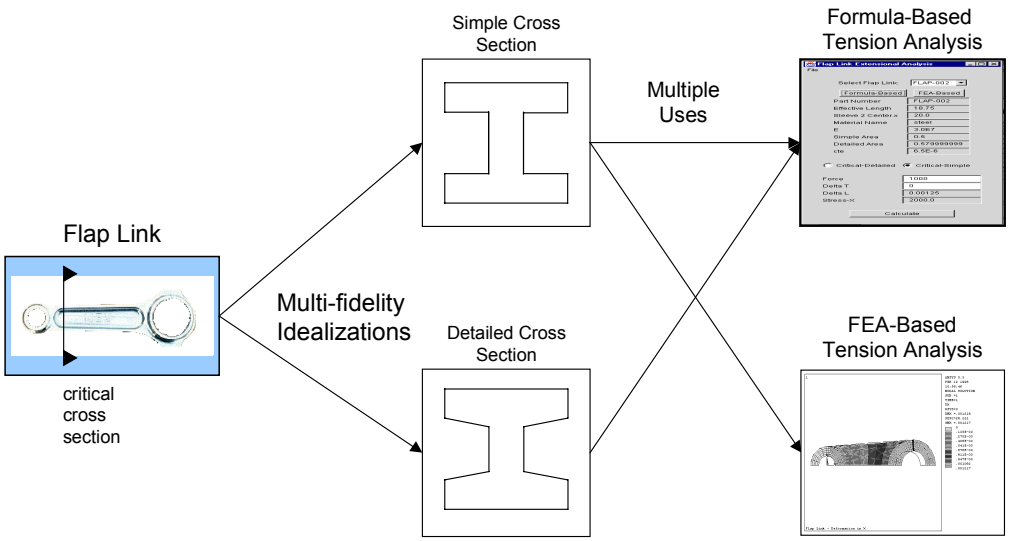


Figure 38-12: Multi-Fidelity Idealizations Example

Secondly, note the *reusability* of the idealizations defined in the APM. As also shown in Figure 38-12, both analyses use the "simple" idealized version of the critical cross section of the shaft.

As indicated by the double-headed arrows in Figure 38-2, the sequence of events just described in this example could also take place in the reverse order. For example, the process could start with the tensional analysis determining a target value for the effective length of the flap link. Then, the same idealization relation used before ("pir1") would be run this time "in reverse" to calculate (or *synthesize*) the coordinates of one of the sleeves given the effective length and the coordinates of the other sleeve as inputs. Once the value for this design attribute is obtained, it could be stored back to the original design repository and read by the solid modeler.

The flap link example used in this subsection will be used throughout the remainder of the chapter to illustrate the APM concepts that will be presented.

APM Representation Overview

As stated in Chapter 27, the main purpose of this thesis is to develop a formal product representation specifically tailored to analysis that facilitates design-analysis integration. The rest of this chapter formally introduces this representation, called the Analyzable Product Model (APM) Representation.

As shown in Figure 38-13, the APM Representation consists of the following four main components, explained in detail in the sections that follow:

1. APM Information Model (Section 41);
2. APM Definition Languages (Section 51);
3. APM Graphical Representations (Section 54); and
4. APM Protocol (Section 58).

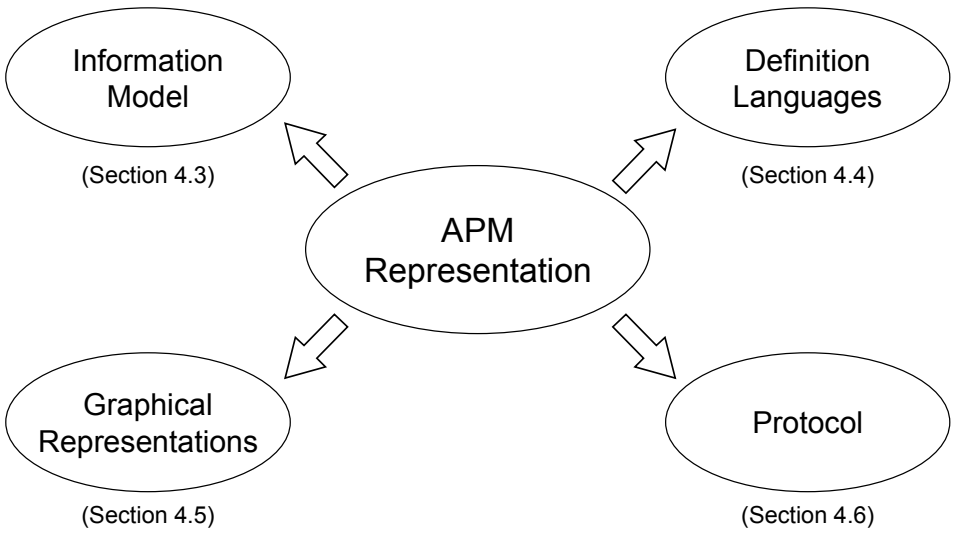


Figure 38-13: APM Representation Components

The *APM Information Model* contains the basic building blocks that make up the theoretical foundation of the APM Representation and provide the basic constructs to build APMs. These constructs describe product information in a way that is particularly convenient for design-analysis integration. Their definitions are presented in this chapter in mathematical

form and therefore are independent from any particular data modeling or programming language.

Two *APM Definition Languages* are introduced in this thesis: the APM Structure Definition Language (APM-S), used to define the *structure* (that is, the source sets, domains, attributes, relations, and source set links) of specific APMs, and the APM Instance Definition Language (APM-I), used to define *instances* of the domains defined in these APMs.

Three *APM Graphical Representations* are also introduced: APM Constraint Schematics Diagrams, APM EXPRESS-G Diagrams and APM Constraint Network Diagrams. The APM Graphical Representations may be used as visual tools for developing new APMs, or for communication and documentation purposes. Each of these graphical representations conveys a certain aspect of the APM better than the others.

The *APM Protocol* is a minimal set of APM properties and conceptual operations for interacting with the APM Representation. These operations can be transformed into programming protocols in specific implementations of the APM Representation, intended to be used by developers of APM-driven applications.

The rest of this chapter introduces the four components of the APM Representation at a conceptual level. As illustrated in Figure 38-14, the APM Information Model and the APM Protocol provide the conceptual basis for deriving general APM properties and characteristics that are implementation-independent. They also provide a specification that can guide implementation in particular computing environments. As also shown in the figure, these two components can be implemented in some target information modeling or programming language (Chapter 64 discusses a prototype implementation of these two components developed by the author). As Figure 38-14 illustrates - and as it will be discussed in detail in Chapter 64 - the constructs defined in the APM Information Model were implemented in this work as EXPRESS entities and as Java classes. The operations of the APM Protocol, on the other hand, were implemented as methods of these Java classes. Chapter 83 describes several test cases that tested and validated the APM Representation against real-world applications using the prototype implementation of the APM presented in Chapter 64.

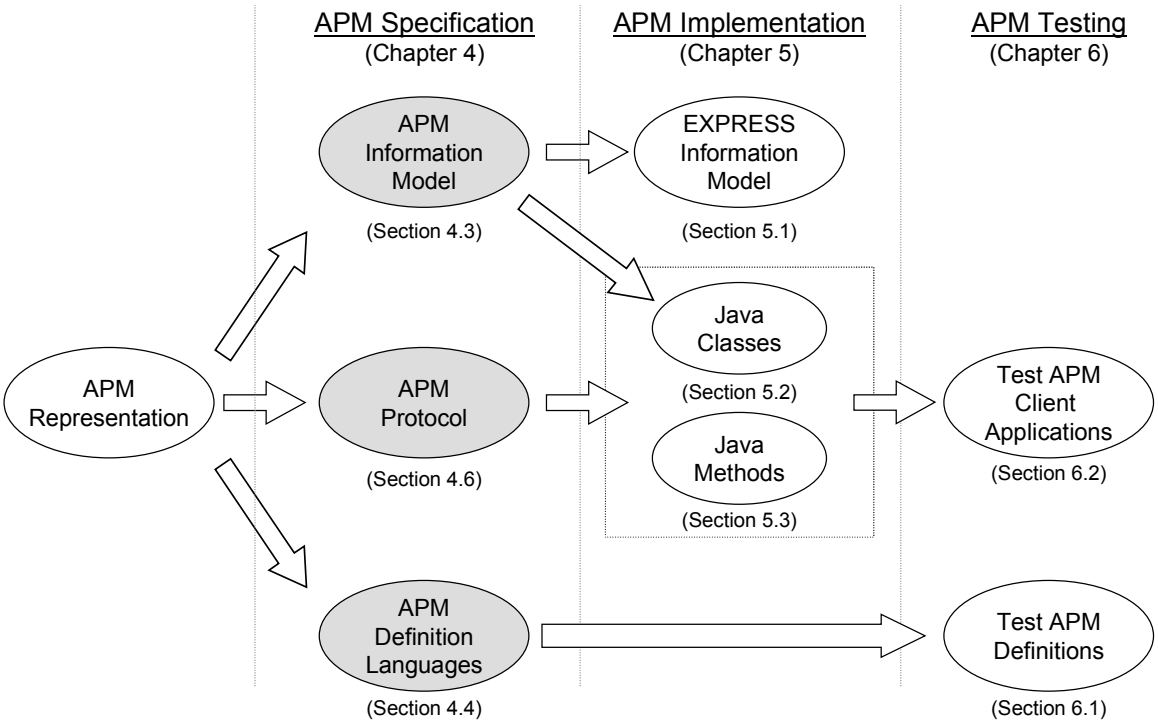


Figure 38-14: APM Representation Implementation and Testing

APM Information Model

This section formally introduces the first and main component of the APM representation: the APM Information Model. The APM Information Model is a formal engineering representation, specifically tailored to analysis, whose primary goal is to facilitate design-analysis integration. It contains the fundamental constructs used to define analyzable product models. It also provides a basis for the rest of the APM Representation components presented in the remaining sections of this chapter.

The fundamental constructs contained in the APM Information Model provide a theoretical foundation for the APM Representation. Their definitions are expressed in terms of set theory notation, and therefore are independent from any particular data modeling or programming language.

As discussed in Chapter 1, design-analysis integration imposes special information requirements that had to be taken into account when developing the APM Information Model. To satisfy these unique requirements of design-analysis integration, the APM Information Model supports information beyond the physical definition of a product, its assembly structure and features. In fact, the detailed physical definition of a product and its geometry is often not as critical for analysis as are the *idealizations* on this physical data. The APM must also be able to contain information that describes how data from multiple sources is to be joined for analysis, and how derived and idealized attributes are obtained from the "real" or "manufacturable" attributes of the product.

Figure 38-15 is a simplified view of the APM Information Model showing its three fundamental groups of constructs - APM Domains, APM Attributes and APM Domain Instances – and how they relate to each other.

As shown in the figure, there are three main types of APM Domains: APM Complex Domains, APM Aggregate Domains and APM Primitive Domains. APM Complex Domains are used to describe the properties of "things" in the physical or in the conceptual world. They contain APM Attributes, which in turn may be APM Complex Attributes, APM Aggregate Attributes or APM Primitive Attributes (not shown in the figure), meaning that their domains are APM Complex Domains, APM Aggregate Domains, or APM Primitive Domains, respectively. APM Complex Attributes also contain attributes, thus allowing for the definition of arbitrarily deep domain-attribute trees. The leaves or terminal nodes of these trees are APM Primitive Attributes, which cannot be subdivided into attributes any further. APM Complex Domains may also contain APM Relations, which describe the mathematical constraints that exist among their terminal attributes.

APM Domain Instances are used to define instances of an APM Domain. There is one subtype of APM Domain Instance corresponding to each subtype of APM Domain.

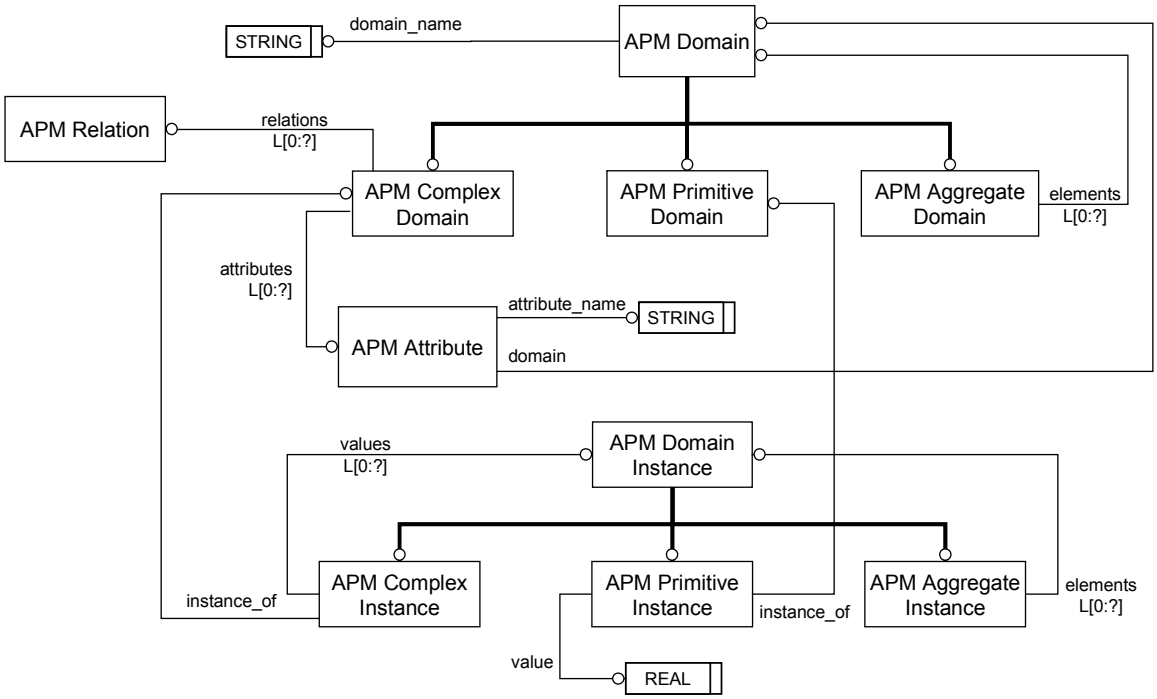


Figure 38-15: Simplified APM Information Model (EXPRESS-G)

APM Domains and their corresponding APM Domain Instances can be grouped in sets called APM Source Sets (Figure 38-16). As it will be discussed later in more detail, APM Source Sets group APM Domains whose instances come from the same source or data repository. As also shown in Figure 38-16, an APM is a collection of these APM Source Sets, plus a list of APM Source Set Links, which specify how instances from different source sets should be joined. The following sections will discuss in greater detail these and other constructs that were omitted for this explanation.

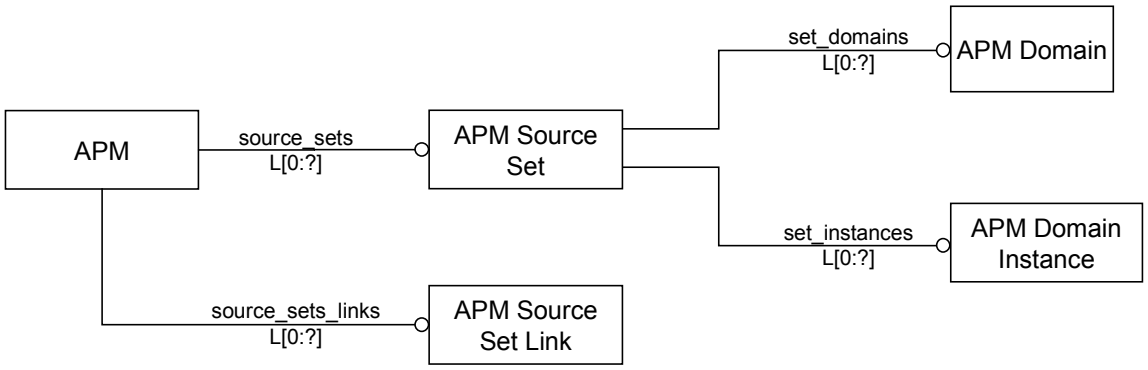


Figure 38-16: Simplified APM Information Model (continued)

Figure 38-17 shows an APM Constraint Schematics Diagram illustrating some of the APM constructs mentioned above (APM Constraint Schematics Diagrams will be introduced in Subsection 56, meanwhile, see Appendix I for a brief summary of their notation). In this type of diagram, circles represent APM Domains and the labeled lines branching out of these circles represent their attributes. On the left side of Figure 38-17 are two APM Source Sets (Source Sets 1 and 2), which contain APM Complex Domains **A** and **X**, respectively. An APM Source Set Link specifies how these two domains are to be joined (specifically, when $\mathbf{A.a_2.b_2}$ is equal to $\mathbf{X.x_1}$) to create the linked APM on the right. The resulting linked APM contains one APM Complex Domain called **A**, which in turn contains four attributes ($\mathbf{a_1}$ through $\mathbf{a_4}$). Attributes $\mathbf{a_1}$, $\mathbf{a_3}$ and $\mathbf{a_4}$ are APM Primitive Attributes ($\mathbf{a_1}$ and $\mathbf{a_4}$ are real numbers and $\mathbf{a_3}$ is a string), whereas attribute $\mathbf{a_2}$ is an APM Complex Attribute of type **B** (an APM Complex Domain), which in turn contains attributes $\mathbf{b_1}$ and $\mathbf{b_2}$. This Subdivision of domains into attributes continues until all the terminal nodes of the tree (namely, attributes $\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{x_1}$, $\mathbf{y_1}$, $\mathbf{y_2}$, $\mathbf{a_3}$ and $\mathbf{a_4}$) are APM Primitive Attributes.

The figure also shows how some of the attributes are related through APM Relations. For example, attributes $\mathbf{a_1}$ and $\mathbf{a_2.b_1}$ are related via relation **R1**. For example, **R1** may specify that the value of $\mathbf{a_1}$ is twice the value of $\mathbf{a_2.b_1}$.

APM Primitive Attributes are grouped into product and idealized. Product APM Primitive Attributes belong to the physical or design description of the product. They are usually defined in one of the original source sets from which the linked APM was built. In this example, attribute $\mathbf{a_1}$ is a Product APM Primitive Attribute (it was originally defined in Source Set 1). Product attributes may be also related to other product attributes via APM

Product Relations (a type of APM Relation) such as **R1**, **R2** and **R3**. Idealized APM Primitive Attributes, on the other hand, belong to the idealized description of the product and are added to the linked APM. In this example, $\mathbf{a}_4$ is the only Idealized APM Primitive Attribute. APM Relations (more specifically APM *Idealization* Relations such as **R4** in the figure) specify how these idealized attributes are obtained from the other attributes of the APM. The figure also illustrates how the constraint network (Subsection 50) can be obtained from the APM.

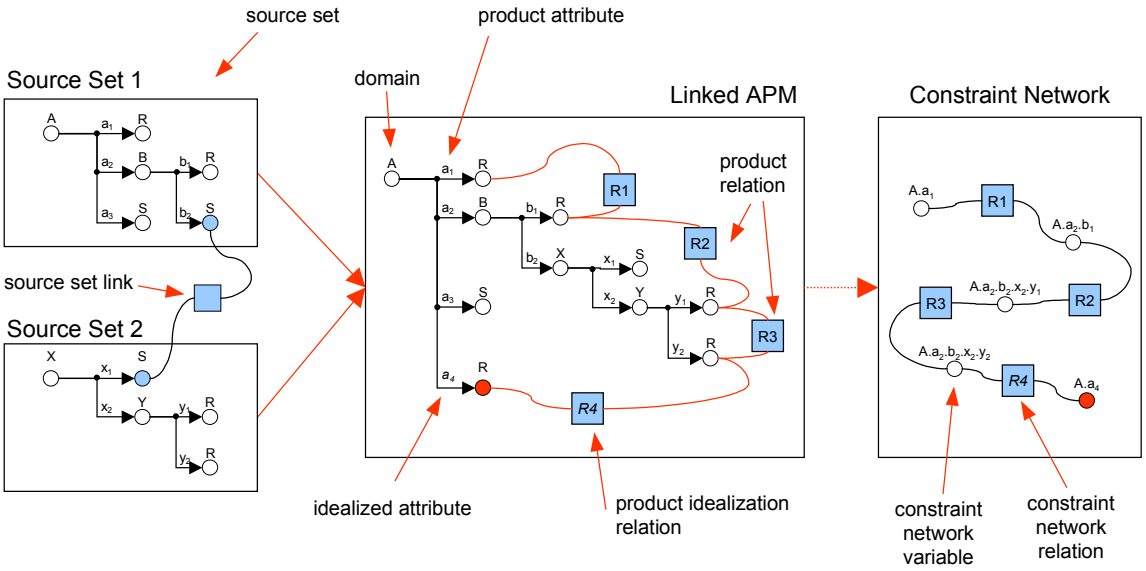


Figure 38-17: Main APM Information Model Constructs

Figure 38-18 shows another APM Constraint Diagram illustrating two sample instances of domain **A** from the linked APM of Figure 38-17. In this case, the relations are:

```
R1 : "a1 == a2.b1 * 2"
R2: "a2.b1 == a2.b2.x2.y1 + 3"
R3: "a2.b2.x2.y1 == -a2.b2.x2.y2"
R4: "a4 == a2.b2.x2.y2^2"
```

In this diagram, circles represent APM Domain Instances. The primitive types (**R** and **S**) have been replaced by actual values (in this example, it is assumed that the values are consistent with the APM Relations). An APM Client Application (Section 89) could access

and manipulate these instances in order to perform some task (such as engineering analysis) using the operations defined in the APM Protocol (Section 58).

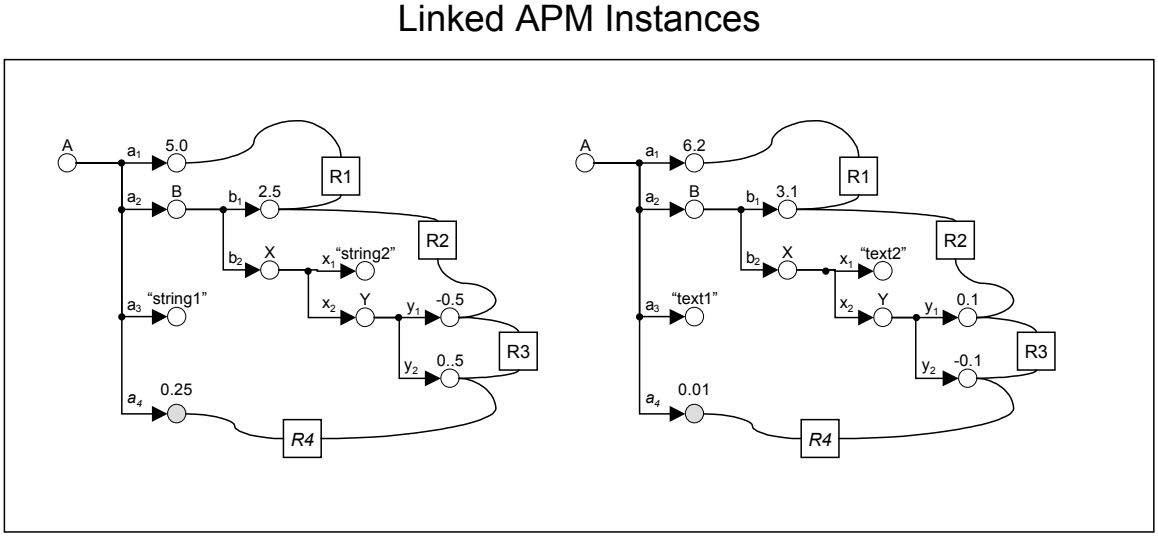


Figure 38-18: Main APM Information Model Constructs (Instances)

An important characteristic of the APM Information Model is that it is *generic*. That is, it is not expressed in terms of any particular domain application. Hence, the APM Information Model effectively becomes a "template" to create domain-specific APMs.

Figure 38-19 illustrates this generic nature of the APM Information Model. At the top of the figure is the generic APM Information Model. Notice from Figure 38-15 that the model is not bound to any particular domain, in other words, its entities ("APM Domain", "APM Attribute", "APM Domain Instance", etc.) may be used to describe potentially anything.

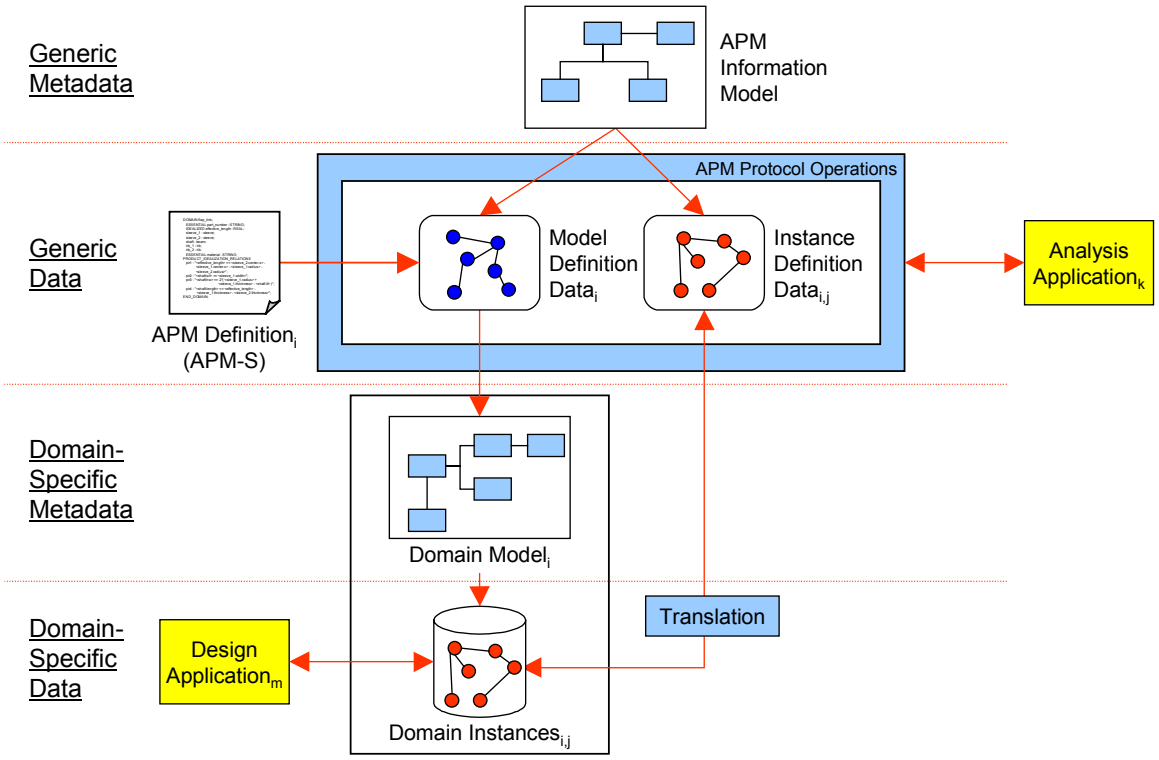


Figure 38-19: Generic Nature of the APM Information Model

The second part of Figure 38-19 shows how the generic constructs of the APM Information Model are populated to obtain the APM Generic Data. Figure 38-20 shows an example of this APM data[10]. In this figure, instances numbered 10 through 80 define a domain called "**plate**" with four attributes: "**length**", "**width**", "**thickness**" and "**hole**". The first three attributes are real numbers, whereas the fourth attribute is of type "**hole**", which in turn has an attribute called "**diameter**" of type real. Instances 100 through 150 define a particular *instance* of the plate domain with **length** = 10.0, **width** = 5.0, **thickness** = 0.5 and a **hole** with **diameter** = 2.5. The first group of instances (instances 10 through 80, labeled "Model Definition Data" in Figure 38-19), define the *structure* of the domain-specific model, whereas the second group of instances (100 through 150, labeled "Instance Definition Data" in Figure 38-19) define the domain-specific *data*. Instances in the second group are instances of the domains defined by the instances in the first group.

[10] STEP Part 21 format is used in this example to define instances of EXPRESS entities. P21 files are discussed in Appendix A.

Model Definition Data

```
#10 = APMObjectDomain( “plate” , ( #20 , #30 , #40 , #50 ) );
#20 = APMAttribute( “length” , #60 );
#30 = APMAttribute( “width” , #60 );
#40 = APMAttribute( “thickness” , #60 );
#50 = APMAttribute( “hole” , #70 );
#60 = APMPrimitiveDomain( “real” );
#70 = APMObjectDomain( “hole” , #80 );
#80 = APMAttribute( “diameter” , #60 );
```

Instance Definition Data

```
#100 = APMObjectInstance( #10 , ( #110 , #120 , #130 , #140 ) );
#110 = APMPrimitiveInstance( #60 , 10.0 );
#120 = APMPrimitiveInstance( #60 , 5.0 );
#130 = APMPrimitiveInstance( #60 , 0.5 );
#140 = APMObjectInstance( #70 , ( #150 ) );
#150 = APMPrimitiveInstance( #60 , 2.5 );
```

Figure 38-20: APM Data Example (STEP P21)

As shown in the third portion of Figure 38-19, model definition instances define a domain-specific model (shown in Figure 38-21 using EXPRESS-G and EXPRESS). The generic instance definition data of Figure 38-20 may be translated to create an equivalent set of domain-specific instances that conforms to this domain-specific model. An example of the result of such translation is shown in Figure 38-22. As illustrated in Figure 38-19, domain-specific instances are normally populated by the design tool.

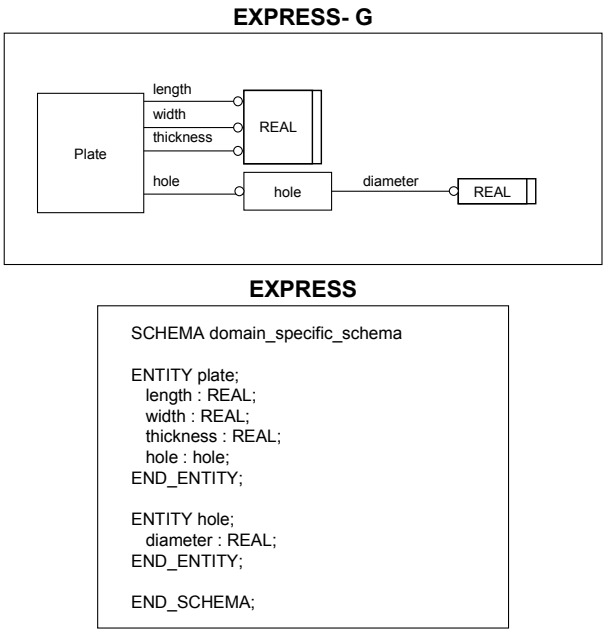


Figure 38-21: Domain-Specific Model Example

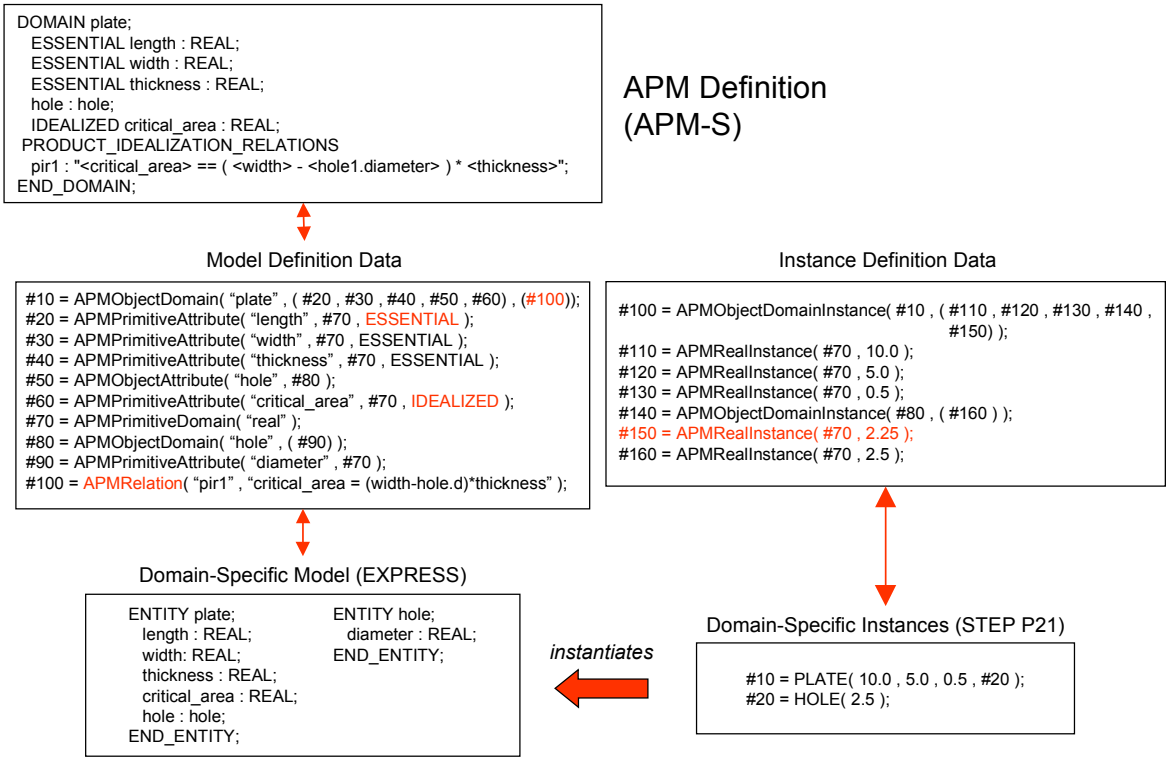


Figure 38-22: Domain-Specific Data Example

Another advantage of having a generic APM Information Model - besides the capability to create specific APMs using the same "template"- is that it allows the definition of *late-bound* operations based on this model. These operations (shown in Figure 38-19 as the box labeled "APM Protocol Operations") are called late-bound because they are designed to access and manipulate APM information without previous knowledge of the structure of the domain-specific entities that will be created. Thus, these operations can be reused in a range of application domains without having to be modified or customized. More importantly, they allow the development of *APM-generic applications*. APM Generic Applications, as it will discussed in more detail in Section 89, are not tied to a particular APM and therefore are designed to work with *any* domain-specific APM. The price to pay for the late-bound nature of these operations is that they require additional overhead, since they must find the data structures and verify that they contain the right attributes, all at run time.

The sections that follow define and discuss the various APM constructs in greater detail, grouped and presented as follows:

1. APM Domains (including multi-level domains*) (Subsection 42);
2. APM Attributes (Subsection 43);
3. APM Domain Instances (Subsection 44);
4. APM Domain Sets and APM Source Sets* (Subsection 45);
5. APM Source Set Links (Subsection 46);
6. Product and Idealized APM Primitive Attributes* (Subsection 47);
7. APM Relations (including idealization relations*) (Subsection 48);
8. Product Model, Manufacturable Product Model* and Analyzable Product Model* (Subsection 49); and
9. Constraint Networks* (Subsection 50).

While some of the definitions that will be presented in these sections are largely based on other general-purpose information modeling languages like EXPRESS (ISO 10303-11 1994; Schenck and Wilson 1994; Wilson 1996), this work adds new terms (such as the ones marked with an asterisk (*) above) that are needed for engineering analysis.

The definitions that follow are given as n-tuples of the form:

$a_i = (x_1 , x_2 , \ldots , x_n)$

Where:

a_i is an APM construct (or structure);

x_i is an attribute of a_i. Attribute x_i may be also a structure or a primitive attribute from $\mathbb{S}$ (the set of strings) or $\mathbb{R}$ (the set of reals);

In addition, a_i defines a set $A = \{ a_1 , a_2 , \ldots , a_m \}$ *intensionally*, by stating the properties that the members of this set must have (Lipschutz 1964; Rosen 1995; Wilson 1996). In other words, these constructs specify "templates" from which specific instances can be created and grouped to form sets.

APM Domains

The APM Information Model defines five types of APM Domains:

1. APM Object Domains (a type of APM Complex Domain);
2. APM Multi-Level Domains (a type of APM Complex Domain);
3. APM Primitive Domains;
4. APM Complex Aggregate Domains (a type of APM Aggregate Domain); and
5. APM Primitive Aggregate Domains (a type of APM Aggregate Domain).

The first two types of APM Domains are also known as ***APM Complex Domains*** (Definition 38-1) because they contain a *list* of attributes (however, as it will be explained in a few paragraphs, the meaning of this list is different in each).

An ***APM Object Domain*** is defined as follows:

$od_i = (domain\_name , \{ a_{s+1} , a_{s+2} , \ldots , a_n \} , \{ r_{p+1} , r_{p+2} , \ldots , r_m \}, supertype\_domain)$

(Definition 38-2)

Where:

od_i : is a specific APM Object Domain;

domain_name is the name of the domain;

$\{ a_{s+1} , a_2 , \ldots , a_n \}$ is the ordered list of local attributes of od_i;

s is the number of all attributes (local and inherited) of *supertype_domain*;

$\{ r_{p+1} , r_2 , \ldots , r_m \}$ is the ordered list of local APM Relations of od_i;

p is the number of all relations (local and inherited) of *supertype_domain*;

supertype_domain is the parent domain of od_i.

domain_name $\in \mathbb{S}$, where $\mathbb{S}$ represents the set of strings[11];

Properties:

$a_j \in \mathcal{A}$, where $\mathcal{A}$ is the set of APM Attributes (Definition 38-28)[12];

$r_k \in \mathcal{R}$, where $\mathcal{R}$ is the set of APM Relations (Definition 38-66);

supertype_domain $\in \mathcal{OD}$, where $\mathcal{OD}$ is defined below (Definition 38-3), or it may also be **null**.

supertype_domain provides the means for defining inheritance hierarchies between APM Object Domains. Following the object-oriented paradigm, a given APM Object Domain od_i inherits the attributes and relations of its parent *supertype_domain*. Thus, in the definition above, $\{ a_{s+1} , a_{s+2} , \ldots , a_n \}$ are called *local attributes*, whereas $\{ a_1 , a_2 , \ldots , a_s \}$ are called *inherited attributes*. Similarly, relations$\{ r_{p+1} , r_{p+2} , \ldots , r_m \}$ are called *local relations*, whereas $\{ r_1 , r_2 , \ldots , r_p \}$ are called *inherited relations*. In this work, a given domain can only have *one* parent domain. In other words, only *simple inheritance* is allowed. The reason for this will be explained in Subsection 66.

---

[11] The convention used hereafter is that all variables of the form xxx_name are strings (that is, xxx_name $\in \mathbb{S}$) unless otherwise noted.

[12] The notation convention used throughout this section is that a_j is an arbitrary member of the set $\{ a_{s+1}, a_{s+2}, \ldots , a_n \}$.

Individual APM Object Domains are grouped to form the ***Set of APM Object Domains***, $\mathcal{OD}$, as follows:

$$\mathcal{OD} = \{ od_1 , od_2 , \ldots , od_n \} \qquad \text{(Definition 38-3)}$$

Most of the "things" in the physical world are described with APM Object Domains. For example, consider the following definition of an APM Object Domain called "Flap Link":

FlapLinkDomain = APMObjectDomain("Flap Link" , { a_1 , a_2 , a_3 , a_4 , a_5 , a_6 })[13]

Where a_1 through a_6 are APM Attributes defined as follows[14]:

a_1 = APMPrimitiveAttribute("part_number" , String , FlapLinkDomain)

("part_number" is the name of the attribute, String is its type, and FlapLinkDomain is the domain of which a_1 is an attribute);

a_2 = APMPrimitiveAttribute("length" , $\mathbb{R}$, FlapLinkDomain);

a_3 = APMObjectAttribute("sleeve_1" , Sleeve , FlapLinkDomain);

a_4 = APMObjectAttribute ("sleeve_2" , Sleeve , FlapLinkDomain);

a_5 = APMObjectAttribute ("shaft" , Beam , FlapLinkDomain); and

a_6 = APMPrimitiveAttribute("material" , $\mathbb{S}$, FlapLinkDomain).

Hence, according to this definition, the following is an instance of domain FlapLinkDomain[15] (assuming that complex objects sleeve5Instance, sleeve6Instance and beam8Instance are valid instances of their respective domains):

link1Instance = APMObjectDomainInstance({"Flap Link-001", 3.4 , sleeve5Instance , sleeve6Instance , beam8 , "steel"} , FlapLinkDomain)

whereas:

---

[13] APMObjectDomain() represents a function that creates APM Object Domains, also known as a ***constructor***.

[14] APM Attribute is formally defined in the next subsection.

[15] APM Object Instance is formally defined later in this section.

link2instance = APMObjectDomainInstance({"Flap Link-002" , 2.5 , sleeve5Instance , beam8Instance } , FlapLinkDomain)

is not a valid instance, because it is missing one of the two sleeves and the material name.

The second type of APM Complex Domain - ***APM Multi-Level Domain*** - is defined as follows:

$$\boldsymbol{md}_i = (\textit{domain\_name} , \{ l_1 , l_2 , \dots , l_n \} , \{ r_1 , r_2 , \dots , r_m \}) \qquad \text{(Definition 38-4)}$$

Where:

$\boldsymbol{md}_i$: is a specific APM Multi-Level Domain;

domain_name is the name of the domain;

$\{ l_1 , l_2 , \dots , l_n \}$ is the ordered list of levels of $\boldsymbol{md}_i$;

$\{ r_1 , r_2 , \dots , r_m \}$ is the ordered list of APM Relations of $\boldsymbol{md}_i$.

Properties:

$l_j \in \mathcal{A}$;

$r_k \in \mathcal{R}$.

Individual APM Multi-Level Domains are grouped to form the ***Set of APM Multi-Level Domains***, $\mathcal{MD}$, as follows:

$$\mathcal{MD} = \{ \boldsymbol{md}_1 , \boldsymbol{md}_2 , \dots , \boldsymbol{md}_n \} \qquad \text{(Definition 38-5)}$$

By looking at the definitions of APM Object Domains (Definition 38-2) and APM Multi-Level Domains (Definition 38-4) it may be noticed that the two are, at least structurally, very similar. However, semantically speaking, the two types of domains are quite different. APM Object Domains are used to describe "things" or "entities" characterized by a list of "attributes" or "features". On the other hand, APM Multi-Level Domains are used to describe things or concepts whose attributes can be grouped into multiple "levels". For

example, as shown in Figure 38-23[16], an object domain **A** has an attribute called $\mathbf{a_1}$. When viewed from one "point of view"[17], $\mathbf{a_1}$ is of type **B** (an object domain with attributes $\mathbf{b_1}$, $\mathbf{b_2}$ and $\mathbf{b_3}$), but when viewed from a different point of view $\mathbf{a_1}$ is of type **C** (a different object domain with attributes $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{c_3}$). These two points of view may be called "levels" of attribute $\mathbf{a_1}$. One possible approach (shown in the second part of Figure 38-23) to capture this multi-level nature of attribute $\mathbf{a_1}$ is to create a new object domain called **D** that groups the attributes from both levels ($\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, $\mathbf{c_1}$, $\mathbf{c_2}$, $\mathbf{c_3}$) and point attribute $\mathbf{a_1}$ of **A** to it. However, with this approach, the fact that these attributes belong to some semantic grouping or level is lost. A better approach (shown in the bottom part of Figure 38-23) is to use a multi-level domain **M**, with two levels (**level1** and **level2** of types **B** and **C**, respectively) and point attribute $\mathbf{a_1}$ of **A** to **M**.

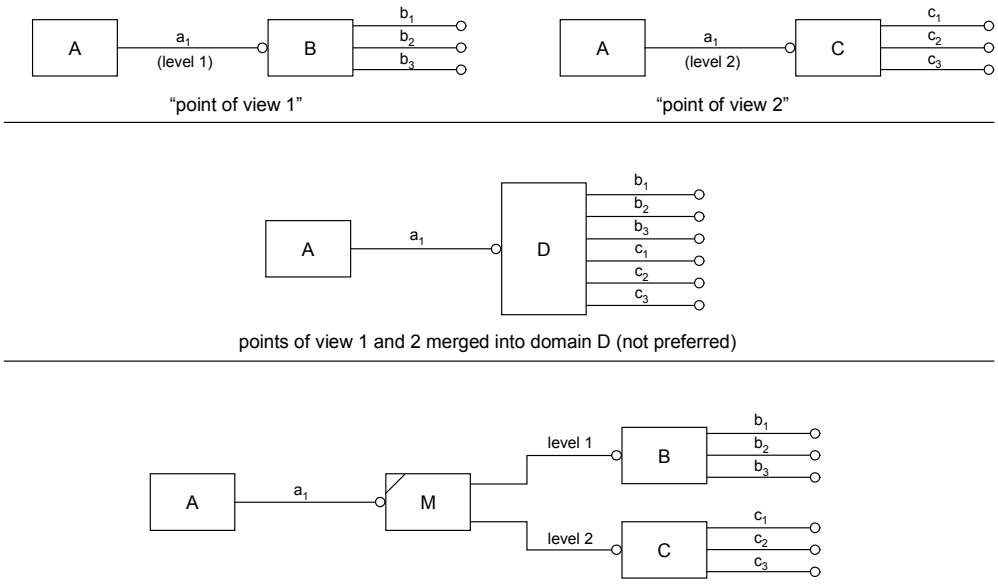


Figure 38-23: Purpose of Multi-Level Domains (Extended EXPRESS-G[18])

[16] The symbols for the data types of the attributes have been omitted in this figure.

[17] The term "point of view" refers to some criterion according to which a product is described. For example, a product could be described from a "thermal" point of view or from a "structural" point of view.

[18] The small diagonal line in the upper-left corner of entity M is not a standard EXPRESS-G symbol. It is introduced in this work to represent multi-level entities.

An important analysis application of APM Multi-Level Domains is to describe features of a product (or the entire product) at different levels of idealization fidelity. This is useful, for example, when an analysis needs to be performed at different levels of precision by using more or less accurate versions or idealizations of the same feature. Thus, a simple analysis may use a simple approximation of the feature, whereas a more precise analysis would use a more detailed version (typically more computationally demanding). The attributes that define the simple and the detailed "versions" of the feature of this example may be grouped into two separate object domains. These two object domains can then be used to define the two levels of the same multi-level domain. In this manner, the two versions or levels of the feature are always available and clearly distinguished, yet grouped together to show that they are semantically related.

A more specific example to better illustrate the utilization of multi-level domain is the following definition of domain `material`[19]:

```
DOMAIN material;
   ESSENTIAL name : STRING;
   stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN material_levels;
   temperature_independent_linear_elastic : linear_elastic_model;
   temperature_dependent_linear_elastic :
      temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;
```

Here, domain `material` has a `name` and two levels of `stress_strain_model`: temperature_independent_linear_elastic and temperature_dependent_linear_elastic (defined in multi-level domain `material_levels`). The attributes that define these two levels are defined in separate object domains (`linear_elastic_model` and `temperature_dependent_linear_elastic_model` not shown above). Thus, for example, the same FR4 material used in a PWB may be modeled in these two ways in different analyses.

According to the definition of multi-level domains above (Definition 38-4), it is also possible to define multi-level domains in which the levels $\{ l_1 , l_2 , \ldots , l_n \}$ are primitive attributes (as

[19] The following is defined using the APM-S language introduced in Subsection 52.

opposed to complex attributes as in the example above). For example, in the following domain:

```
DOMAIN flap_link;
   ESSENTIAL part_number : STRING;
   IDEALIZED effective_length : MULTI_LEVEL effective_length_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN effective_length_levels;
   torsional : REAL;
   extensional : REAL;
END_MULTI_LEVEL_DOMAIN;
```

Attribute `effective_length` may take two values (that is, calculated differently) depending on the type of analysis being run: `torsional` and `extensional` (both `REAL`). Thus, if `aFlapLinkInstance` were an instance of domain `flap_link`, the value of its effective length accessed by a torsional analysis would be:

```
aFlapLinkInstance.effective_length.torsional
```

and by an extensional analysis:

```
aFlapLinkInstance.effective_length.extensional
```

The sets of APM Object Domains and APM Multi-Level Domains are joined to form the ***Set of APM Complex Domains***, $\mathcal{CD}$, as follows:

$$\mathcal{CD} = \mathcal{OD} \; i \; \mathcal{MD} \qquad \text{(Definition 38-6)}$$

The third type of APM Domain, ***APM Primitive Domain***, is defined as:

$$pd_i = (\ domain\_name\ ) \qquad \text{(Definition 38-7)}$$

Where:

pd_i: is a specific APM Primitive Domain;

domain_name is the name of the primitive domain.

Properties:

domain_name ∈ { "REAL" , "STRING" }.

From the definition above it can be concluded that, effectively, only two APM Primitive Domains are defined in this work: the domain of the real numbers ($\mathbb{R}$), and domain of the strings ($\mathbb{S}$)[20].

Individual APM Primitive Domains are grouped to form the ***Set of APM Primitive Domains***, $\mathcal{PD}$, as follows:

$$\mathcal{PD} = \{ pd_1 , pd_2 , \ldots , pd_n \} \qquad \text{(Definition 38-8)}$$

Or, considering that $\mathbb{R}$ and $\mathbb{S}$ are the only primitive domains defined in this work:

$$\mathcal{PD} = \{ \mathbb{R} , \mathbb{S} \} \qquad \text{(Definition 38-9)}$$

In general, instances of APM Primitive Domains are atomic units of data (in other words, their elements are *not* given as n-tuples, but just as single numbers or strings). They are also known as *terminal* or *simple* domains, since they are not further subdivided into attributes. All domains will eventually wind up being composed of these simple domains.

APM Aggregate Domains (which include the fourth and fifth types of APM Domains) define a template to create aggregates of objects[21]. The only information needed to specify a template to create aggregates is the name of the domain and the domain of its elements. There are two types of APM Aggregate Domains, depending on the types of its elements: APM Complex Aggregate Domains, and APM Primitive Attribute Domains.

An ***APM Complex Aggregate Domain*** is defined as follows:

$$cad_i = (domain\_name , domain\_of\_elements) \qquad \text{(Definition 38-10)}$$

Where:

cad_i: is a specific APM Complex Aggregate Domain;

domain_name is the name of the aggregate domain;

---

[20] More primitive domains could be included in the definition of APM Primitive Domain (an obvious example of a primitive type not being included is the Integers) and handled in a similar manner. However, for simplicity, only real numbers and strings will be considered in this work. These two types satisfy most of the needs of engineering analysis.

[21] For this work, the only type of aggregate that will be described is *List* (an ordered collection). Other possible types of aggregates include *bags*, *sets* and *arrays*.

domain_of_elements is the parent domain of the elements of the aggregate.

Properties:

domain_of_elements $\in \mathcal{CD}$.

Notice that the distinguishing characteristic of APM Complex Aggregate Domains is that the domains of its elements are APM Complex Domains. The domains of the elements need not be the same; the only requirement is that they be subtypes of *domain_of_elements*.

Individual APM Complex Aggregate Domains are grouped to form the ***Set of APM Complex Aggregate Domains***, $\mathcal{CAD}$, as follows:

$$\mathcal{CAD} = \{ cad_1 , cad_2 , \ldots , cad_n \} \qquad \text{(Definition 38-11)}$$

To illustrate the use of APM Complex Aggregate Domains, consider the Printed Wiring Board (PWB) example shown in Figure 38-24. PWBs are made up of several layers, and each layer has a thickness and a material. An object domain called "PWB" may be defined having an attribute called "layup". The domain of this attribute is an APM Complex Aggregate Domain called "ListOfLayers", and the domain of its elements is "Layer". The definition for ListOfLayers is: ("ListOfLayers" , Layer), where Layer (the domain of the elements) is an APM Object Domain with two attributes, thickness (a Real) and material (a String).

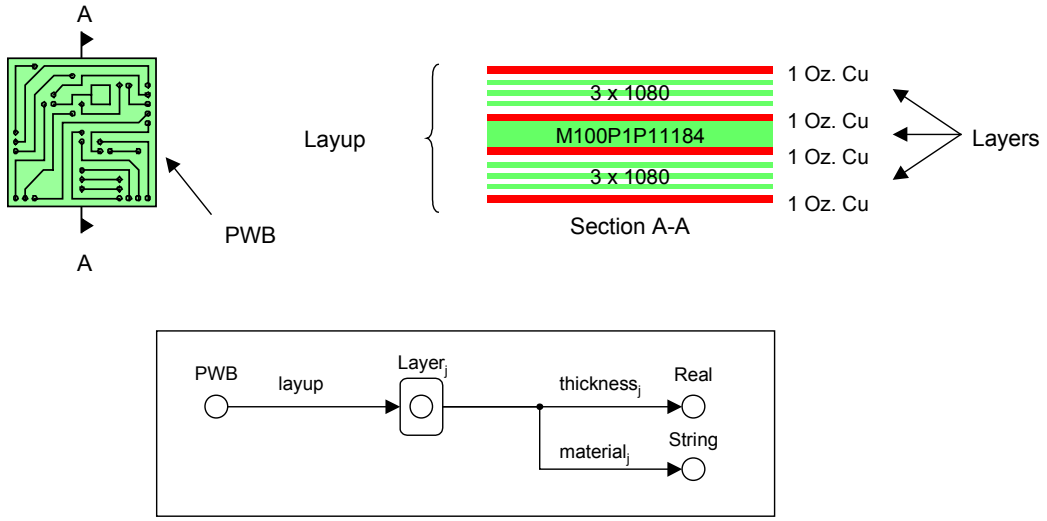


Figure 38-24: APM Complex Aggregate Domain Usage Example

The second type of APM Aggregate Domain, ***APM Primitive Aggregate Domain***, is defined as follows:

$pad_i = (domain\_name , domain\_of\_elements)$ (Definition 38-12)

Where:

pad_i: is a specific APM Primitive Aggregate Domain;

domain_name is the name of the aggregate domain;

domain_of_elements is the domain of the elements of the aggregate.

Properties:

$domain\_of\_elements \in \mathcal{PD}$.

Notice that the distinguishing characteristic of APM Primitive Aggregate Domains is that the domain of its elements is an APM Primitive Domain.

Individual APM Primitive Aggregate Domains are grouped to form the ***Set of APM Primitive Aggregate Domains***, $\mathcal{PAD}$, as follows:

$\mathcal{PAD} = \{ pad_1 , pad_2 , \ldots , pad_n \}$ (Definition 38-13)

An example of an APM Primitive Aggregate Domain is:

TemperatureMeasurements = APM PrimitiveAggregateDomain("Temperature-Measurements" , $\mathbb{R}$)

Which defines an aggregate called "TemperatureMeasurements" whose elements are real numbers. A valid instance of domain TemperatureMeasurements could be:

temperatureMeasurements = (125.00 , 132.50 , 145.15, 110.75)

Notice that, in both types of APM Aggregate Domains, *domain_of_elements* is either an APM Complex Domain (in APM Complex Aggregate Domains) or an APM Primitive Domain (in APM Primitive Aggregate Domains). In other words, *domain_of_elements* cannot be *another* APM Aggregate Domain. This means that aggregates of aggregates (for example, a List of Lists of FlapLinkages) cannot be defined. This constraint greatly simplifies the implementation of some of the APM operations that will be presented later in this chapter. In addition, aggregates of aggregates do not appear often, and when they do, it is still possible to define them using intermediate complex domains. For example, Figure 38-25 shows an APM Domain **X** with an attribute $\mathbf{x_1}$, which is a list of lists of **B**s. This can be replaced by making $\mathbf{x_1}$ a list of **A'**s each **A'** having an attribute $\mathbf{a_1}$ which is a list of **B**s.

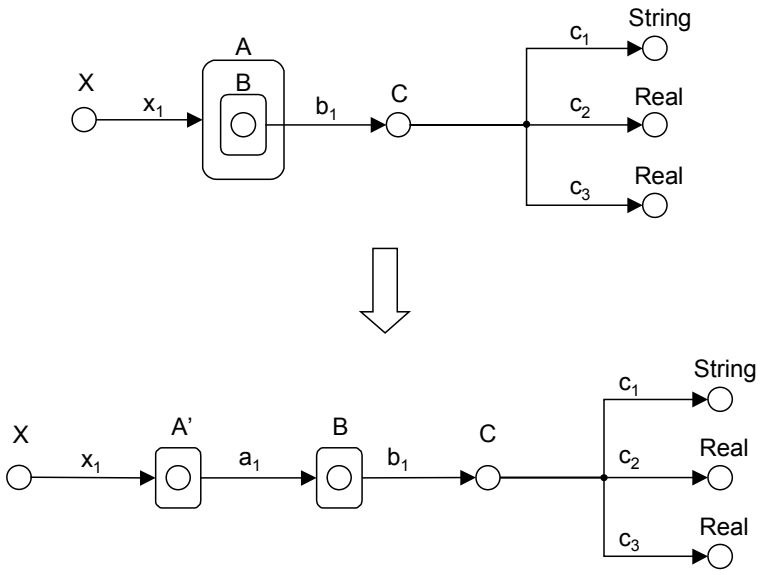


Figure 38-25: Using Intermediate APM Object Domains to Replace Aggregates of Aggregates.

The sets of APM Complex Aggregate Domains and APM Primitive Aggregate Domains are joined to form the ***Set of APM Aggregate Domains***, $\mathcal{AD}$, as follows:

$$\mathcal{AD} = \mathcal{CAD} \; i \; \mathcal{PAD} \qquad \text{(Definition 38-14)}$$

Finally, the sets of APM Complex Domains, APM Primitive Domains and APM Aggregate Domains (that is, all the sets of APM Domains defined so far) are joined to form the ***Set of APM Domains***, $\mathcal{D}$, as follows:

$$\mathcal{D} = \mathcal{CD} \; i \; \mathcal{PD} \; i \; \mathcal{AD} \qquad \text{(Definition 38-15)}$$

APM Attributes

As mentioned in the previous subsection, APM Attributes are used to define APM Complex Domains (see Definitions 38-2 and 38-4). There are five types of APM Attributes, according to their domains:

1. APM Object Attributes;
2. APM Multi-Level Attributes;

3. APM Primitive Attributes[22];
4. APM Primitive Aggregate Attributes; and
5. APM Complex Aggregate Attributes.

The first type of APM Attribute, ***APM Object Attribute***, is defined as follows:

oa_i = (*attribute_name* , *domain* , *container_domain*) (Definition 38-16)

Where:

oa_i: is a specific APM Object Attribute;

attribute_name is the name of the attribute;

domain is the domain of the attribute;

container_domain is the domain containing the attribute.

Properties:

domain $\in \mathcal{OD}$;

container_domain $\in \mathcal{CD}$.

Individual APM Object Attributes are grouped to form the ***Set of APM Object Attributes***, $\mathcal{OA}$, as follows:

$\mathcal{OA} = \{ oa_1 , oa_2 , \ldots , oa_n \}$ (Definition 38-17)

The second type of APM Attribute, ***APM Multi-Level Attribute***, is defined as:

ma_i= (*attribute_name* , *domain* , *container_domain*) (Definition 38-18)

Where:

ma_i: is a specific APM Multi-Level Attribute;

[22] APM Primitive Attributes will be subdivided into Product (Essential and Reduntant) and Idealized Attributes (Subsection 47)

attribute_name is the name of the attribute;

domain is the domain of the attribute;

container_domain is the domain containing the attribute.

Properties:

$domain \in \mathcal{MD}$;

$container\_domain \in \mathcal{CD}$.

Individual APM Multi-Level Attributes are grouped to form the ***Set of APM Multi-Level Attributes***, $\mathcal{MA}$, as follows:

$\mathcal{MA} = \{ ma_1 , ma_2 , \ldots , ma_n \}$ (Definition 38-19)

The third type of APM Attribute, ***APM Primitive Attribute***, is defined as:

$pa_i = (attribute\_name , domain , container\_domain)$ (Definition 38-20)

Where:

pa_i: is a specific APM Primitive Attribute;

attribute_name is the name of the attribute;

domain is the domain of the attribute;

container_domain is the domain containing the attribute.

Properties:

$domain \in \mathcal{PD}$;

$container\_domain \in \mathcal{CD}$.

Individual APM Primitive Attributes are grouped to form the ***Set of APM Primitive Attributes***, $\mathcal{PA}$, as follows:

$\mathcal{PA} = \{ pa_1 , pa_2 , \ldots , pa_n \}$ (Definition 38-21)

In the same way as APM Primitive Domains are also known as terminal and simple domains, APM Primitive Attributes are also known as terminal or simple attributes because they do not get decomposed further.

The fourth type of APM Attribute, ***APM Primitive Aggregate Attribute***, is defined as:

$paa_i = ($ *attribute_name* , *domain* , *container_domain*, *lower_size_bound* , *upper_size_bound*$)$

(Definition 38-22)

Where:

paa_i: is a specific Primitive Aggregate Attribute;

attribute_name is the name of the attribute;

domain is the domain of the attribute;

container_domain is the domain containing the attribute;

lower_size_bound is the minimum number of elements;

upper_size_bound is the maximum number of elements.

Properties:

domain $\in \mathcal{PAD}$;

container_domain $\in \mathcal{CD}$;

lower_size_bound $\in \mathbb{S}$;

upper_size_bound $\in \mathbb{S}$.

Individual APM Primitive Aggregate Attributes are grouped to form the ***Set of APM Primitive Aggregate Attributes***, $\mathcal{PAA}$, as follows:

$\mathcal{PAA} = \{ paa_1 , paa_2 , \ldots , paa_n \}$ (Definition 38-23)

The fifth and last type of APM Attribute, ***APM Complex Aggregate Attribute***, is defined as:

caa_i = (*attribute_name* , *domain* , *container_domain*, *lower_size_bound* , *upper_size_bound*)

(Definition 38-24)

Where:

caa_i: is the set of APM Complex Aggregate Attribute;

attribute_name is the name of the attribute;

domain is the domain of the attribute;

container_domain is the domain containing the attribute;

lower_size_bound is the minimum number of elements;

upper_size_bound is the maximum number of elements.

Properties:

domain $\in \mathcal{CAD}$;

container_domain $\in \mathcal{CD}$;

lower_size_bound $\in \mathbb{S}$;

upper_size_bound $\in \mathbb{S}$.

Individual APM Complex Aggregate Attributes are grouped to form the ***Set of APM Complex Aggregate Attributes***, $\mathcal{CAA}$, as follows:

$\mathcal{CAA} = \{ caa_1 , caa_2 , \ldots , caa_n \}$ (Definition 38-25)

In general, as Definitions 38-16, 38-18, 38-20, 38-22, and 38-24 indicate, the information required to define an APM Attribute is the attribute name, the domain of the attribute and the domain that contains the attribute. Although the domain of the attribute may vary

(depending on the type of APM Attribute), attributes can only be defined inside APM Complex Domains.

For APM Aggregate Attributes (Definitions 38-22 and 38-24), the size bounds (minimum and maximum number of elements) of the aggregate must also be defined. Strings are used to specify these bounds in order to be able to use some non-numeric symbol to indicate "many" or "unknown" number of elements (a question mark - "?" - is used in this work).

The sets of APM Object Attributes and APM Multi-Level Attributes are joined to form the ***Set of APM Complex Attributes***, $\mathcal{CA}$, as follows:

$$\mathcal{CA} = \mathcal{OA} \; \dot{\cup} \; \mathcal{MA} \qquad \text{(Definition 38-26)}$$

Also, the sets of APM Complex Aggregate Attributes and APM Primitive Aggregate Attributes are joined to form the ***Set of APM Aggregate Attributes***, $\mathcal{AA}$, as follows:

$$\mathcal{AA} = \mathcal{PAA} \; \dot{\cup} \; \mathcal{CAA} \qquad \text{(Definition 38-27)}$$

Finally, the ***Set of APM Attributes***, $\mathcal{A}$, can be defined as follows:

$$\mathcal{A} = \mathcal{CA} \; \dot{\cup} \; \mathcal{PA} \; \dot{\cup} \; \mathcal{AA} \qquad \text{(Definition 38-28)}$$

It is useful to define the membership relationship "***belongs to***" ($\in^*$) between APM Attributes and APM Complex Domains. In the case of APM Object Domains, this relationship can be recursively defined as follows:[23]

$$a_i \in^* od_j \text{ iff } a_i \in \{ a_1 , a_2 , \dots , a_n \} \qquad \text{(Definition 38-29)}$$

In other words, an attribute a_i belongs to an object domain od_j if a_i is an attribute (local or inherited) of od_j. When this is true, *container_domain* of a_i is equal to od_j. Similarly, for APM Multi-Level Domains:

$$a_i \in^* md_j \text{ iff } a_i \in \{ l_1 , l_2 , \dots , l_n \} \qquad \text{(Definition 38-30)}$$

In other words, an attribute a_i belongs to a multi-level domain md_j if a_i is a level of md_j. When this is true, *container_domain* of a_i is equal to md_j.

[23] This and all the relationships defined in this chapter are summarized in Appendix B for future reference.

The asterisk in $\in^*$ is used to distinguish between this relationship and the traditional definition of set membership ($\in$). The statement $a_i \in cd_j$ would incorrectly imply that an attribute a_i *is an element* of the set defined by complex domain cd_j. Instead, $a_i \in^* cd_j$ is used to state that attribute a_i *belongs to* complex domain cd_j, in other words, that a_i is one of the attributes (or levels) of cd_j.

The relationship defined in Definition 38-29 may also be called ***direct belonging***, because a_i is *directly* one of the attributes (or levels) of cd_j. Extending this definition further, an ***indirect belonging relationship***, called $\in^\sim$, may be defined recursively as:

$a_i \in^\sim cd_j$ iff $a_i \in^* cd_k$, where cd_k is the domain of a_k and ($a_k \in^* cd_j$ ∨ $a_k \in^\sim cd_j$)

(Definition 38-31)

In other words, an attribute a_i *indirectly belongs* to a complex domain cd_j if it directly belongs to a complex domain cd_k, which is the domain of an attribute a_k, which directly or indirectly belongs to cd_j.

A consequence of Definition 38-31 is that direct belonging implies indirect belonging. In other words:

$a_i \in^* cd_j \Rightarrow a_i \in^\sim cd_j$.

Membership relationships analogous to the ones defined in Definitions 38-29 and 38-31 may also be defined between two APM Attributes as follows:

$a_i \in^* a_j$ iff $a_i \in^* cd_j$, where cd_j is the domain of a_j. (Definition 38-32)

In other words, an attribute a_i *directly belongs* to another attribute a_j if a_i directly belongs to the domain of a_j.

Similarly, the indirect belonging relationship ($\in^\sim$) between two APM Attributes is defined as:

$a_i \in^\sim a_j$ iff $a_i \in^\sim cd_j$, where cd_j is the domain of a_j. (Definition 38-33)

In other words, an attribute a_i *indirectly belongs* to another attribute a_j if a_i indirectly belongs to the domain of a_j.

The member relationships presented above can also be illustrated using the dot convention commonly used in object-oriented programming to indicate "part of" relations. For example, if **a.b.c** is an attribute of domain **D**, the following can be stated:

a $\in^{*}$ **D** (attribute **a** directly belongs to domain **D**)

a, b, **c** $\in^{\sim}$ **D** (attributes **a**, **b** and **c** indirectly belong to domain **D**)

b $\in^{*}$ **a** (attribute **b** directly belongs to attribute **a**)

c $\in^{*}$ **b** (attribute **c** directly belongs to attribute **b**)

c $\in^{\sim}$ **a** (attribute **c** indirectly belongs to attribute **a**)

APM Domain Instances

An APM Domain Instance is simply a *particular instance* of a given APM Domain. There are five types of APM Domain Instances, according to the domains of which they are instances:

1. APM Object Domain Instances;
2. APM Multi-Level Domain Instances;
3. APM Primitive Domain Instances;
4. APM Primitive Aggregate Domain Instances; and
5. APM Complex Aggregate Domain Instances.

The first type of APM Domain Instance, ***APM Object Domain Instance*** is defined as:

$oi_i = (\{ i_1 , i_2 , \ldots , i_n \} , domain)$ (Definition 38-34)

Where:

oi_i: is a particular instance of *domain*;

domain is the APM Object Domain being instantiated;

$\{ i_1 , i_2 , \ldots , i_n \}$ is the ordered list of all attribute instances of oi_i;

Properties:

$oi_i \in^i$ *domain*, where the "is an instance of" relationship ($\in^i$) is defined in Definition 38-35;

domain $\in \mathcal{OD}$;

$i_j \in^i d_j$, where d_j is the domain of the j^{th} attribute of *domain*.

$i_j \in \mathcal{I} \setminus i_j \in^i \boldsymbol{d}_j$, where $\mathcal{I}$ is defined in Definition 38-51 and $\boldsymbol{d}_j$ is the domain of a_j;

$a_j \in \mathcal{A}$;

$a_j \in^* domain$.

In other words, an APM Object Domain Instance contains a list of instances; each instance in this list corresponding to an attribute of the domain being instantiated. These instances are, in turn, APM Domain Instances as well.

The "is an instance of" relationship between APM Domain Instances and APM Domains (denoted $\in^i$) is defined as follows:

> Definition 38-35: *An APM Domain Instance* i_i ***is an instance of*** *an APM Domain* $\boldsymbol{d}_j$ *(denoted* $i_i \in^i \boldsymbol{d}_j$*) when the domain of* i_i *equals* $\boldsymbol{d}_j$.

Individual APM Object Domain Instances are grouped to form the ***Set of APM Object Domain Instances***, $\mathcal{OI}$, as follows:

$\mathcal{OI} = \{ oi_1 , oi_2 , \ldots , oi_n \}$ (Definition 38-36)

Notice that the definition of APM Object Domain Instance above (Definition 38-34) contains more information than just a n-tuple of values of the form $(i_1 , i_2 , \dots , i_n)$ as it also contains a reference to *domain*. An n-tuple would only contain the values of the attributes of an APM Object Domain, but it would not contain any reference to the APM Object Domain itself. Thus, there would be no way to perform any type checking on the values, because there is not a reference to the "template" that was used to create them.

In the flap link example of Subsection 42, link1Instance is an *instance* of the APM Object Domain "FlapLinkDomain".

The second type of APM Domain Instance, ***APM Multi-Level Domain Instance***, is defined as follows:

$mi_i = (\{ i_1 , i_2 , \dots , i_n \} , domain)$ (Definition 38-37)

Where:

mi_i: is a particular instance of *domain*;

domain is the APM Multi-Level Domain being instantiated;

$\{ i_1 , i_2 , \dots , i_n \}$ is the ordered list of level instances of mi_i;

Properties:

$mi_i \in^{i} domain$;

$i_j \in^{i} d_j$, where d_j is the domain of the j^{th} level of *domain*;

$domain \in \mathcal{MD}$;

$i_j \in \mathcal{I} \setminus i_j \in^{i} d_j$, where d_j is the domain of l_j;

$l_j \in \mathcal{A}$;

$l_j \in^{*} domain$.

The definition above is similar to the definition of APM Object Domain Instances (Definition 38-34). The difference is that each instance i_j in the list of instances corresponds to a *level* of the domain being instantiated, whereas in APM Object Domain Instances it corresponds to an *attribute*.

Individual APM Multi-Level Domain Instances are grouped to form the ***Set of APM Multi-Level Domain Instances***, $\mathcal{MI}$, as follows:

$$\mathcal{MI} = \{ mi_1 , mi_2 , \ldots , mi_n \} \qquad \text{(Definition 38-38)}$$

The sets of APM Object Domain Instances and APM Multi-Level Domain Instances are joined to form the ***Set of APM Complex Domain Instances***, $\mathcal{CI}$, as follows:

$$\mathcal{CI} = \mathcal{OI} \ i \ \mathcal{MI} \qquad \text{(Definition 38-39)}$$

The third type of APM Domain Instance, ***APM Primitive Domain Instance***, is defined as follows:

$$pi_i = (v , domain) \qquad \text{(Definition 38-40)}$$

Where:

pi_i: is a particular instance of *domain*;

v is the value of the instance;

domain is the APM Primitive Domain being instantiated.

Properties:

$pi_i \in^i domain$;

$v \in \{ \mathbb{R} \ i \ \mathbb{S} \}$;

$domain \in \mathcal{PD}$.

In other words, an APM Primitive Domain Instance contains an atomic value v (a real or a string). For this reason, APM Primitive Domain Instances are also known as *terminal* or

simple instances APM Primitive Instances are particularly important because they contain the values that can actually be used to populate an analysis model.

Individual APM Primitive Domain Instances are grouped to form the ***Set of APM Primitive Domain Instances***, $\mathcal{PI}$, as follows:

$$\mathcal{PI} = \{ pi_1 , pi_2 , \dots , pi_n \} \quad \text{(Definition 38-41)}$$

More specifically, when *domain* is $\mathbb{R}$, the APM Primitive Domain Instances is called APM Real Instance, and when *domain* is $\mathbb{S}$, it is called APM String Instance. Thus, an ***APM Real Instance*** is defined as:

$$ri_i = (v , \mathbb{R}) \quad \text{(Definition 38-42)}$$

Individual APM Real Instances are grouped to form the ***Set of APM Real Instances***, $\mathcal{RI}$ as follows:

$$\mathcal{RI} = \{ ri_1 , ri_2 , \dots , ri_n \} \quad \text{(Definition 38-43)}$$

Likewise, an ***APM String Instance*** is defined as:

$$si_i = (v , \mathbb{S}) \quad \text{(Definition 38-44)}$$

Individual APM String Instances are grouped to form the ***Set of APM String Instances, $\mathcal{SI}$***, as follows:

$$\mathcal{SI} = \{ si_1 , si_2 , \dots , si_n \} \quad \text{(Definition 38-45)}$$

The fourth type of APM Domain Instance, ***APM Primitive Aggregate Domain Instance***, is defined as follows:

$$pai_i = (\{ pi_1 , pi_2 , \dots , pi_n \} , domain) \quad \text{(Definition 38-46)}$$

Where:

pai_i: is a particular instance of *domain*;

domain is the APM Primitive Aggregate Domain being instantiated;

$\{ pi_1 , pi_2 , \dots , pi_n \}$ is the ordered list of primitive element instances of pai_i.

Properties:

$pai_i \in^{i} domain$;

$domain \in \mathcal{PAD}$;

$pi_j \in \mathcal{PI}$ and $pi_j \in^{i} pd_i$, where pi_j is the j^{th} element of the aggregate and pd_i is the domain of the elements *domain_of_elements*, an APM Primitive Domain, defined in *domain* (see Definition 38-12).

Individual APM Primitive Aggregate Domain Instances are grouped to form the ***Set of APM Primitive Aggregate Domain Instances***, $\mathcal{PAI}$, as follows:

$$\mathcal{PAI} = \{ pai_1 , pai_2 , \ldots , pai_n \} \qquad \text{(Definition 38-47)}$$

The fifth and last type of APM Domain Instance, ***APM Complex Aggregate Domain Instance***, is defined as follows:

$$cai_i = (\{ ci_1 , ci_2 , \ldots , ci_n \} , domain) \qquad \text{(Definition 38-48)}$$

Where:

cai_i: is a particular instance of *domain*;

domain is the common parent APM Complex Aggregate Domain being instantiated;

$\{ ci_1 , ci_2 , \ldots , ci_n \}$ is the ordered list of complex element instances of cai_i.

Properties:

$cai_i \in^{i} domain$;

$domain \in \mathcal{CAD}$;

$ci_j \in \mathcal{CI}$ and $ci_j \in^{i} cd_i$, where ci_j is the j^{th} element of the aggregate and cd_i is the domain of the elements *domain_of_elements*, an APM Complex Domain, defined in *domain* (see Definition 38-10).

Note that, in the definition above, the elements in $\{ ci_1 , ci_2 , \ldots , ci_n \}$ may be instances of different domains, as long as they are all subtypes of *domain* - the common parent.

Individual APM Complex Aggregate Domain Instances are grouped to form the ***Set of APM Complex Aggregate Domain Instances***, $\mathcal{CAI}$, as follows:

$$\mathcal{CAI} = \{ cai_1 , cai_2 , \ldots , cai_n \} \qquad \text{(Definition 38-49)}$$

The sets of APM Primitive Aggregate Domain Instances and APM Complex Aggregate Domain Instances are joined to form the ***Set of APM Aggregate Domain Instances***, $\mathcal{AI}$, as follows:

$$\mathcal{AI} = \mathcal{PAI} \; i \; \mathcal{CAI} \qquad \text{(Definition 38-50)}$$

Finally, the ***Set of APM Domain Instances***, $\mathcal{I}$, is defined as:

$$\mathcal{I} = \mathcal{CI} \; i \; \mathcal{PI} \; i \; \mathcal{AI} \qquad \text{(Definition 38-51)}$$

APM Domain Sets and APM Source Sets

An APM Domain Set provides a way to group APM Domains according to any arbitrary criterion. An ***APM Domain Set*** is defined as follows:

$$s_i = (domain\_set\_name , \{ d_1 , d_2 , \ldots , d_n \}) \qquad \text{(Definition 38-52)}$$

Where:

s_i : is a specific domain set;

domain_set_name is the name of the domain set;

$\{ d_1 , d_2 , \ldots , d_n \}$ is the set of domains that belong to this set.

Properties:

$d_j \in \mathcal{D}$, where d_j is the j^{th} APM Domain of the domain set.

When an APM Domain Set is populated with instances it is called an APM Domain Set Instance. An ***APM Domain Set Instance*** is defined as follows:

$$si_i = (s_j , \{ i_1 , i_2 , \dots , i_n \}) \qquad \text{(Definition 38-53)}$$

Where:

si_i : is a specific domain set instance;

s_j : is a specific domain set;

$\{ i_1 , i_2 , \dots , i_n \}$ is the set of instances contained in this source set instance.

$i_k \in^i cd_q$ and $cd_q \in^* s_j$, where the $\in^*$ relationship between a domain and a domain set is defined in Definition 38-56.

Notice that, by definition, the instances in $\{ i_1 , i_2 , \dots , i_n \}$ can only be instances of APM *Complex* Domains. The reason is that APM Complex Domain Instances are the only ones that can exist independently (that is, not inside another instance) within the present APM framework.

A special type of APM Domain Sets, called APM Source Sets, is defined as follows:

> Definition 38-54: *An **APM Source Set** is an APM Domain Set whose domains* $\{ d_1 , d_2 , \dots , d_n \}$ *are populated with instances coming from the same source or data repository.*

In other words:

> *Given* $si_i = (s_j , \{ i_1 , i_2 , \dots , i_n \})$, if s_j *is an APM Source Set then* $\{ i_1 , i_2 , \dots , i_n \}$ *come from the same source or data repository.*

For example, two source sets could be defined for a given product: one grouping the domains that define the geometry of the product, and the second grouping the domains that define materials and their properties. Instances of the domains in the first set could come from a data file created by a solid modeler, whereas instances of the domains in the second set could come from a materials database.

Individual APM Source Sets are grouped to form the ***Set of APM Source Sets***, $\mathcal{S}$, as follows:

$\mathcal{S} = \{ s_1 , s_2 , \ldots , s_n \}$ (Definition 38-55)

Finally, it is useful to define the membership relationship ***belongs to*** ($\in^*$) between APM Domains and APM Domain Sets as:

$d_i \in^* s_j$ iff $d_i \in \{ d_1 , d_2 , \ldots , d_n \}$ (Definition 38-56)

In other words, a domain d_i belongs to a domain set s_j if d_i is one of the member domains of s_j.

One explicit restriction regarding domain set membership that is placed in this APM representation is that a given d_i cannot belong to more than one domain set in the *same* APM. More formally:

if $d_i \in^* s_j$ and $d_i \in^* s_k$, and

if $s_j \in^* apm_x^{(p)}$ and $s_k \in^* apm_y^{(p)}$

then $apm_x^{(p)} \neq apm_y^{(p)}$

APM Source Set Links

APM Source Set Links specify when and how instances from different source sets in the same APM should be joined (or "linked") in order to obtain a single set of instances. An ***APM Source Set Link*** between two APM Source Sets - set_1 and set_2 - is defined as follows:

$l_i = (name , set_1 , set_2 , key\_attribute_1 , key\_attribute_2 , link\_condition , insertion\_attribute , inserted\_attribute)$ (Definition 38-57)

Where:

l_i : is a specific APM Source Set Link;

name is a name for the set link;

set_1 is the name of the first source set;

set_2 is the name of the second source set;

$key\_attribute_1$ is the key attribute in set_1;

$key\_attribute_2$ is the key attribute in set_2;

link_condition is a Boolean proposition;

insertion_attribute is the attribute in set_1 that is going to be replaced by *inserted_attribute* from set_2;

inserted_attribute is the attribute in set_2 that is going to replace *insertion_attribute* in set_1.

Properties:

$name, set_1, set_2 \in \mathcal{S}$;

$set_1 \surd set_2$;

$set_1 , set_2 \in^* apm_i^{(p)}$ (see definition of $\in^*$ between APMSourceSets and APMs in Definition 38-74 and definition of $apm_i^{(p)}$ in Definition 38-69);

$(key\_attribute_1 \in \mathcal{PA}) \setminus (key\_attribute_1 \in \tilde{} cd_1) \setminus (cd_1 \in^* set_1)$;

$(key\_attribute_2 \in \mathcal{PA}) \setminus (key\_attribute_2 \in \tilde{} cd_2) \setminus (cd_2 \in^* set_2)$;

link_condition returns true or false and may or may not involve $key\_attribute_1$ and/or $key\_attribute_2$;

$insertion\_attribute \in a_1 , a_1 \in \tilde{} cd_3 , cd_3 \in^* set_1$;

$inserted\_attribute \in a_2 , a_2 \in \tilde{} cd_4 , cd_4 \in^* set_2$.

In other words, an APM Source Set Link is defined by specifying the following information: two different source sets (belonging to the same APM), a key attribute in each set, a link condition, an insertion attribute in the first set and an inserted attribute from the second set. The key attributes must be primitive attributes that belong to the sets being linked. The

insertion attribute is an APM Attribute that indirectly belongs to a complex domain that belongs to the first set. The inserted attribute is an APM Attribute that indirectly belongs to a complex domain that belongs to the second set. When the link condition is true, $key\_attribute_1$ in set_1 is replaced by $key\_attribute_2$ in set_2. Although the link condition may be, in general, any Boolean proposition, for this work it is limited to a proposition of the form:

$$key\_attribute_1\ logical\_operator\ key\_attribute_2$$

Where *logical_operator* may be equal ("=="), greater than (">"), greater or equal than (">="), less that ("<"), less or equal than ("<="), or not equal ("!="). More complex link conditions could be defined between primitive values, which could involve if-then rules or arbitrary algorithmic procedures, but only propositions of the form above will be considered for this work.

It is important to point out that by defining when and how the insertion and the inserted attributes are linked at the domain level, APM Source Set Links effectively specify when and how *instances* of these domains will be linked.

To illustrate how APM Source Set Links work, consider the example illustrated in Figure 38-26. This example shows two source sets $\mathbf{S_1}$ and $\mathbf{S_2}$. $\mathbf{S_1}$ contains one domain called **A** and $\mathbf{S_2}$ contains one domain called **X**. A source set link is defined between the two sets as:

$$\text{SetLink1} = \text{APMSourceSetLink}(\text{ "link1" }, S_1 , S_2 , a_2 , x_1 , a_2 == x_1 , a_1 , x_2)$$

This definition specifies that if the strings of $\mathbf{a_2}$ and $\mathbf{x_1}$ are equal, then $\mathbf{a_1}$ should be replaced with $\mathbf{x_2}$. The result of this link is also shown in the figure. Notice how the domain of attribute $\mathbf{a_1}$ changed from **B** to **Y**. The name of the attribute ($\mathbf{a_1}$) is still the same.

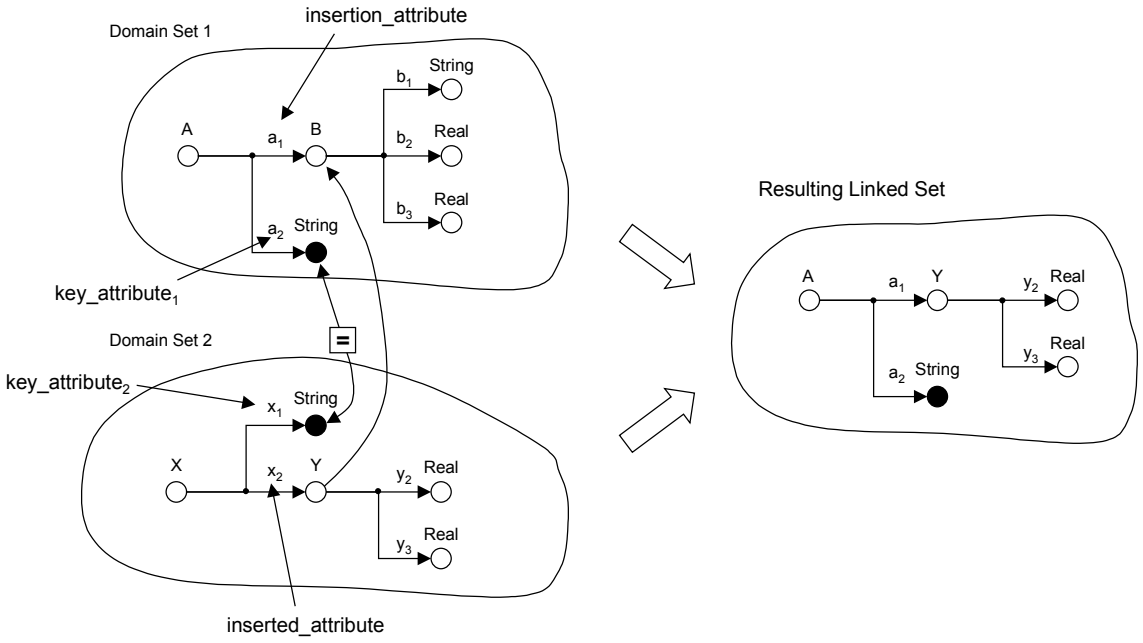


Figure 38-26: APM Source Set Link Example with Undesired Information Loss

Notice that **B** and its attributes (**b_1**, **b_2**, and **b_3**) are lost as a result of this link because **Y** overlaps it when the two sets are linked. In general, this is an undesirable consequence because information is *lost* during the linking process, and, in principle, if that information was defined in the set in the first place was because it is required for analysis. This loss of information can be avoided by adding the following conditions to the original definition of source set link (Definition 38-57):

$insertion\_attribute = key\_attribute_1$;

$key\_attribute_2 \in \tilde{} \; inserted\_attribute$;

$inserted\_attribute \in ca_2, ca_2 \in \tilde{} \; cd_4, cd_4 \in^* set_2$.

The first condition (making $key\_attribute_1$ equal to the *insertion_attribute*) ensures that the attribute that is being replaced (the *insertion_attribute*) points exclusively to the key value (a string or a real) and not to any other information. The second condition (ensuring that the *inserted_attribute* contains $key\_attribute_2$) ensures that the key attribute is not lost during

the linking. In this case, as shown in the third condition above, *inserted_attribute* must be a complex attribute in order to be able to contain $key\_attribute_2$.

With these constraints, *insertion_attribute* does not need to be specified (since it is equal to $key\_attribute_1$). Therefore, Definition 38-57 can be simplified and rewritten as:

$$\mathcal{I}_i' = (\ name\ ,\ set_1\ ,\ set_2\ ,\ key\_attribute_1\ ,\ key\_attribute_2\ ,\ link\_condition\ ,\ inserted\_attribute\ )$$

(Definition 38-58)

Where:

$\mathcal{I}_i'$: is a simplified APM Source Set Link (Type 1).

An example of this constrained type of source set link is shown in Figure 38-27. In the example in this figure, the link is defined as:

$$\text{SetLink2} = \text{APMSourceSetLink}(\text{“link2”}\ ,\ S_1\ ,\ S_2\ ,\ b_1\ ,\ z_1\ ,\ b_1 == z_1\ ,\ x_1\ )$$

Notice that in this example $\mathbf{b_1}$ is both $key\_attribute_1$ and *insertion_attribute*. Also notice that the *inserted_attribute* ($\mathbf{x_1}$) indirectly contains $key\_attribute_2$ ($\mathbf{z_1}$). Thus, in the linked version, the value referred to by $\mathbf{b_1}$ is maintained as $\mathbf{z_1}$.

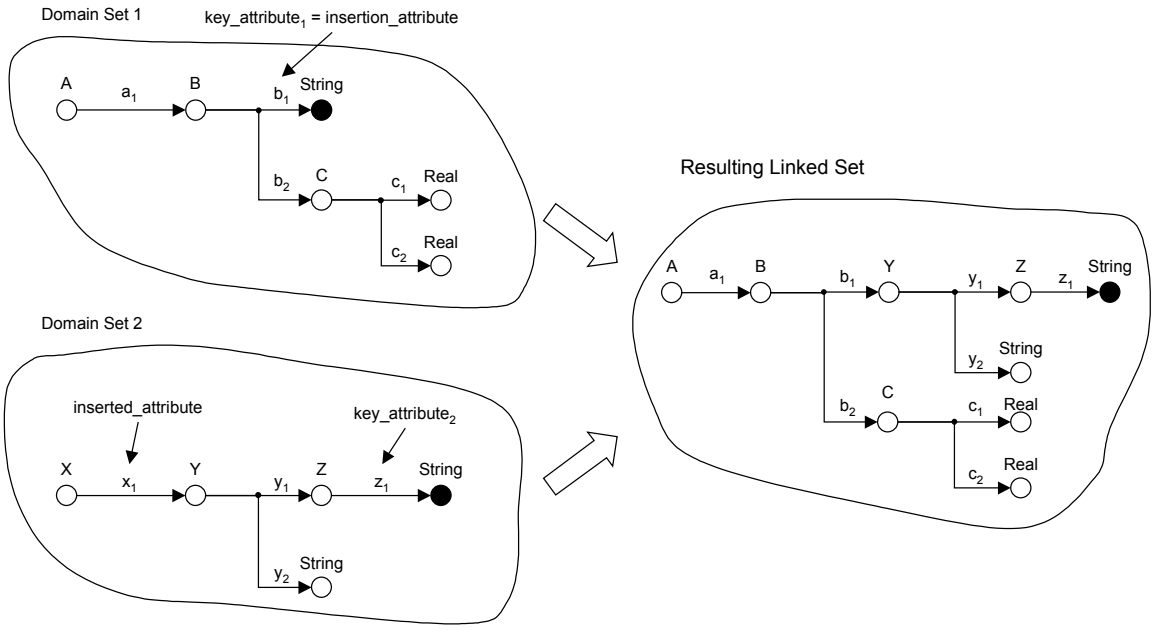


Figure 38-27: Example of a Type 1 Simplified Source Set Link

The definition of a source set link can be further simplified by adding the following condition:

$$key\_attribute_2 \in^* inserted\_attribute$$

In other words, *inserted_attribute* must directly contains $key\_attribute_2$. In this way, *inserted_attribute* does not need to be specified since it will be, by default, the attribute that contains $key\_attribute_2$. With this additional condition, the definition of a source set link finally becomes:

$$\mathcal{I}_i'' = (name , key\_attribute_1 , key\_attribute_2 , link\_condition) \qquad \text{(Definition 38-59)}$$

Where:

$\mathcal{I}_i''$: is a simplified APM Source Set Link (Type 2).

Figure 38-28 shows the same example of Figure 38-27 this time using the simplification of Definition 38-59. Notice that this time, since $key\_attribute_2$ is equal to *inserted_attribute*, **Z** replaces the string to which attribute $\mathbf{b_1}$ was pointing.

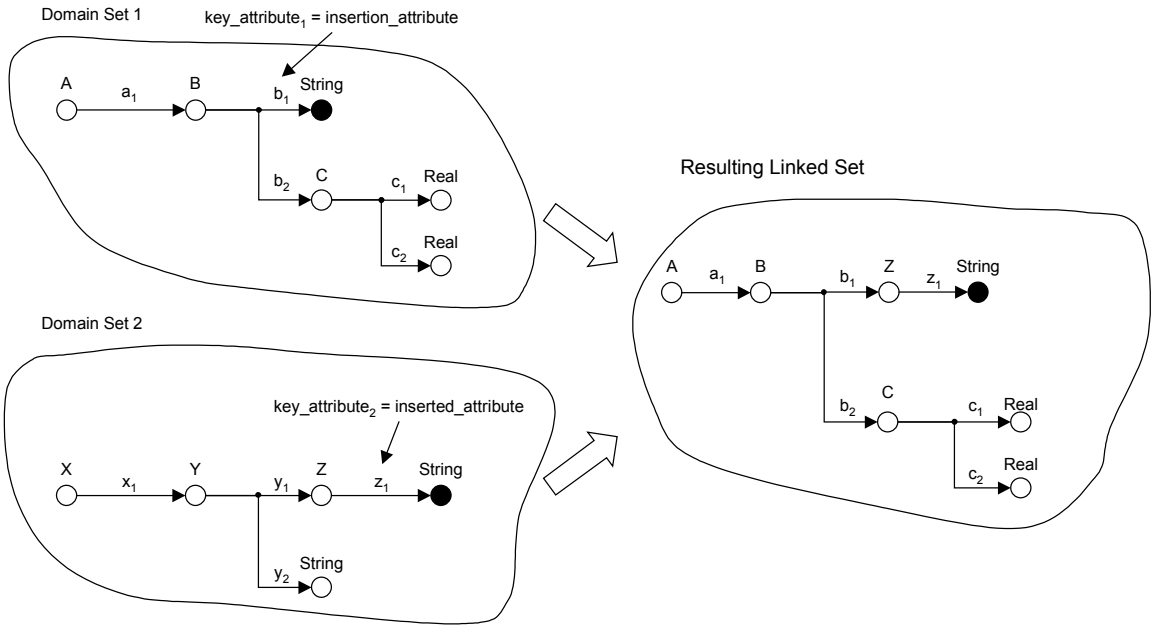


Figure 38-28: Example of a Type 2 Simplified Source Set Link

A more realistic example of an application of a Type 2 simplified source set link is shown in Figure 38-29 (simplified from the APM definition of Figure 38-7). In this example there are two source sets: one containing information about the geometry of the flap link (`flap_link_geometric_model`) and the second corresponding to a database of material properties (`flap_link_material_properties`). The two source sets are linked through attribute `material` of domain `flap_link` in the first source set and attribute `name` of domain `material` in the second source set. After linking, attribute `material` of domain `flap_link` – which originally pointed to a `String` – now points to the complex domain `material`.

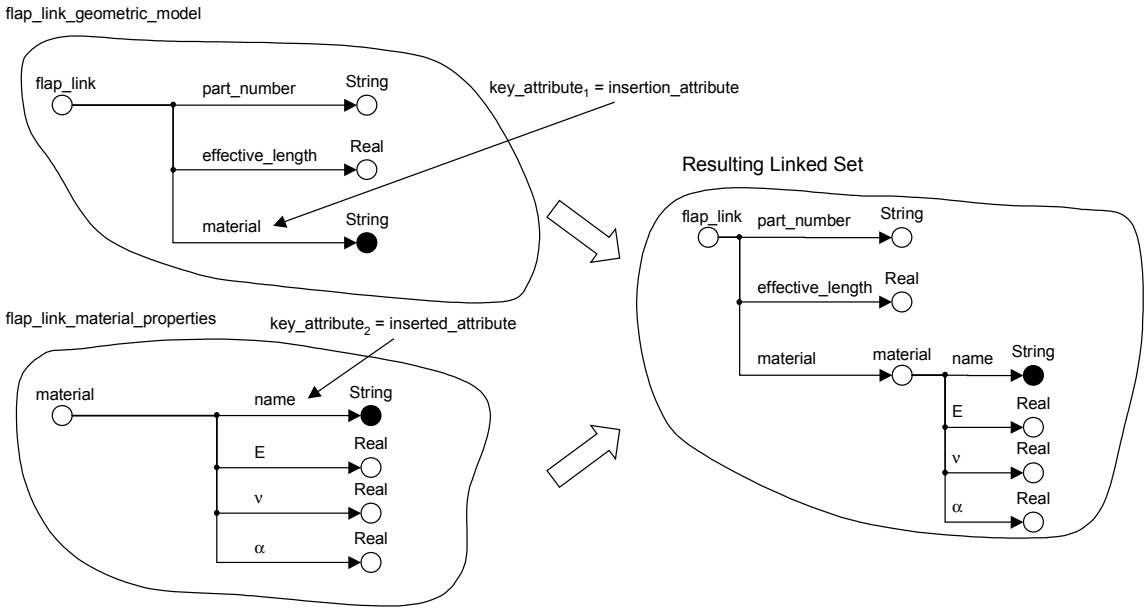


Figure 38-29: A More Realistic Example of a Type 2 Simplified Source Set Link

Finally, individual APM Source Set Links are grouped to form the ***Set of APM Source Set Links***, $\mathcal{L}$, as follows (considering only the type 2 simplified definition of source set link):

$$\mathcal{L} = \{ \mathcal{I}_1'', \mathcal{I}_2'', \ldots, \mathcal{I}_n'' \} \qquad \text{(Definition 38-60)}$$

Product and Idealized APM Primitive Attributes

APM Primitive Attributes may be grouped into two categories: Product Attributes and Idealized Attributes. In this work, this categorization is reserved exclusively for APM *Primitive* Attributes, since primitive attributes are the only ones that actually hold a value and therefore can be used directly as inputs to manufacturing or analysis models. It might be useful to define entire *domains* as product or idealized, but this capability will be left as a future extension to the APM Representation (Chapter 110). For the purposes of this work, a given domain may be considered idealized if all its primitive attributes are idealized.

APM Product Attributes are defined as follows:

> Definition 38-61: *An APM Primitive Attribute is an* ***APM Product Attribute*** *if it belongs to the physical or design description of the product.*

APM Product Attributes can usually be measured with an instrument directly on the product or perceived by the senses. Examples of product attributes are the length of a plate, the diameter of a hole, the weight of a part, the coordinate of a point, and the distance between two features.

More specifically, APM *Essential* Product Attributes are defined as follows:

> Definition 38-62: *An APM Product Attribute is an* ***APM Essential Product Attribute*** *if it is part of the set of minimum and necessary attributes to manufacture the part*

As it will be described in more detail later in this section (Section 49), APM Essential Product Attributes make up the "manufacturable" description of the product.

APM Redundant Product Attributes are defined as follows:

> Definition 38-63: *An APM Product Attribute is an* ***APM Redundant Product Attribute*** *if it is not an APM Essential Product Attribute.*

The decision of which attributes are essential and which redundant is, in general, arbitrary. For example, if a part has three lengths $\mathbf{L_1}$, $\mathbf{L_2}$ and $\mathbf{L_{total}}$, and these lengths are related by the expression "$\mathbf{L_{total} = L_1 + L_2}$", then any two of the three lengths may be chosen as essential and the third will automatically be redundant. Despite being redundant, APM Redundant Product Attributes are often defined to add expressiveness to the description of the product.

Finally, APM Idealized Attributes are defined as follows:

> Definition 38-64: *an APM Primitive Attribute is an* ***APM Idealized Attribute*** *if it belongs to the idealized description of the product.*

Idealized Attributes are *fictitious* in nature. In other words, these attributes are "made up" by the analyst, based on his or her experience, and they usually cannot be physically measured on the product, since they do not actually exist. In general, idealized attributes are the result of simplifying or transforming product attributes using heuristic knowledge. Examples of Idealized Attributes are the "critical" area of a plate, the "effective" length of a link, and the "lumped" coefficient of thermal expansion of a multi-layer PWB. Idealized Attributes are often needed for analysis because most analysis models are expressed mathematically in

terms of idealized attributes of the product (or in terms of a combination of product *and* idealized attributes).

Figure 38-30 is a Venn diagram showing the relationship between product and idealized attributes. Two properties of this diagram of particular significance are:

1. *An APM Primitive Attribute is either a Product Attribute or an Idealized Attribute; and*
2. *A Product Attribute is either Essential or Redundant.*

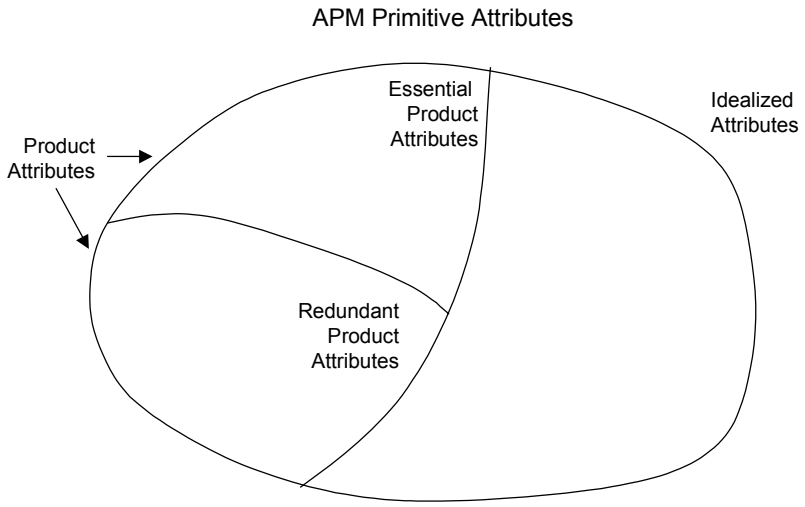


Figure 38-30: Relationship between Product and Idealized Attributes

As mentioned earlier (Section 39), an important characteristic of APM Primitive Attributes is that they are *intrinsic* to the part. Intrinsic attributes can be viewed as attributes that *inherently belong* to the part and therefore are independent of the environment in which the part exists[24]. Given the same part, their values are normally invariant. For example, the effective length and the radius of sleeve 1 are intrinsic attributes of the flap link. According to this definition, temperature-dependant properties such as the coefficient of thermal

[24] Strictly speaking, these attributes (mostly geometric and material properties) are "intrinsic" with respect to a reference environment (typically a load-free, standard temperature and pressure, etc.). The main point is that infinite derivations from this standard environment are possible, therefore those derivations and their corresponding changes in the part are derivable from the intrinsic prperties contained in the APM.

expansion are not intrinsic (their values vary with temperature). However, the coefficients used in the mathematical models that predict their values are intrinsic, because they belong to the material of which the part is made.

Other types of attributes that participate in analysis and that are *not* defined in APMs are *environmental attributes* and *behavioral attributes*. Environmental attributes describe the environment in which the part performs at a given point in time as well as its boundary conditions. Applied loads and temperatures fall in this category. Behavioral attributes describe the behavior of the part when it is subjected to these environmental conditions. They are normally considered the outputs of the analysis models. Common examples of behavioral attributes are deformations and stresses.

APM Relations

APM Relations define how APM Primitive Attributes are mathematically related. An ***APM Relation*** is defined as follows:

$r_i = (\ relation\_name\ ,\ \{\ pa_1\ ,\ pa_2\ ,\ \dots\ ,\ pa_n\ \}\ ,\ relation\ )$ (Definition 38-65)

Where:

r_i: a specific APM Relation;

relation_name is the name of the relation;

$\{\ pa_1\ ,\ pa_2\ ,\ \dots\ ,\ pa_n\ \}$ is the ordered list of related primitive attributes;

relation is a mathematical expression involving the related attributes.

Properties:

$pa_j \in \mathcal{PA}$;

$relation \in \mathbb{S}$.

Individual APM Relations are grouped to form the ***Set of APM Relations***, $\mathcal{R}$, as follows:

$\mathcal{R} = \{\ r_1\ ,\ r_2\ ,\ \dots\ ,\ r_n\ \}$ (Definition 38-66)

As indicated in Definitions 38-2 and 38-4, APM Relations are used as part of the definition of APM Complex Domains. They are defined in the APM Complex Domain that contains, directly or indirectly, all the attributes involved in the relation.

It is important to notice that APM Relations only define mathematical relationships between *primitive* attributes. In addition, they do not define or imply in any way which of these attributes are inputs and which are outputs.

Some useful relations involving aggregates can be defined. These relations use operations on *whole aggregates* (such as Sum, Max , Min , Average, etc.) instead of the values of *individual elements* of the aggregate. To illustrate this type of operations, consider the two domains **A** and **B** shown in Figure 38-31.

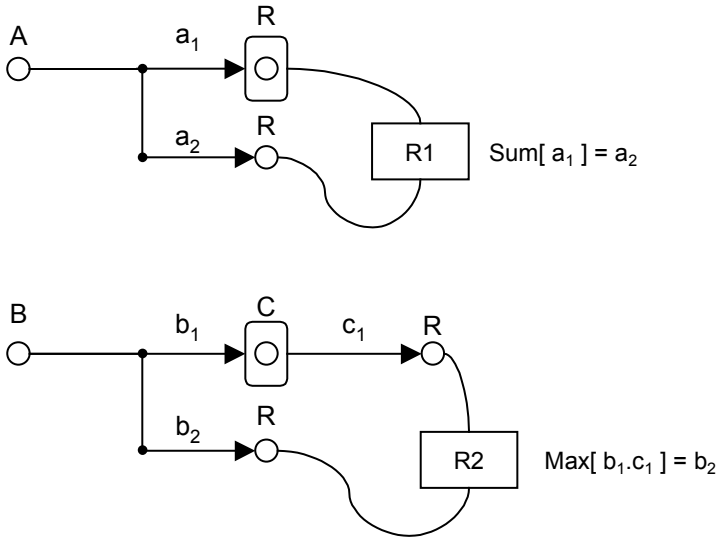


Figure 38-31: Aggregate Relations Example

In this example, relation **R1** is defined as follows:

R1 = ("R1" , { a_1 , a_2 } , "Sum[a_1] = a_2")

This relation states the $\mathbf{a_2}$ must be equal to the *sum* of the values in aggregate $\mathbf{a_1}$. An instance of **A** now could be:

instanceOfA = APMObjectDomainInstance({ { 2.3 , 4.4 , 5.5 , 3.2 } , 14.4 } , A)

Where 14.4 is the sum of the elements in { 2.3 , 4.4 , 5.5 , 3.2 }. Likewise, for domain **B**, relation **R2** is defined as follows:

R2 = ("R2" , { $b_1.c_1$, b_2 } , "Max[$b_1.c_1$] = b_2")

Which states that $\mathbf{b_2}$ is equal to the maximum value of the first attributes of the complex elements in $\mathbf{b_1}$. For example, an instance of **B** now could be:

instanceOfB = APMObjectDomainInstance ({ { 1.0 , 2.0 , 3.0 } , { 10.0 , 5.0 , 6.0 } , { -4.0 , 8.0 , 9.0 } } , 10.0 } , B)

Where 10.0 is the maximum of 1.0 , 10.0 and –4.0.

APM Relations can be divided into APM Product Relations and APM Product Idealization Relations. APM Product Relations are defined as:

Definition 38-67: *An APM Relation is an* ***APM Product Relation*** *if all the primitive attributes in the list of related attributes* { pa_1 , pa_2 , … , pa_n } *are APM Product Attributes.*

And APM Product Idealization Relations are defined as:

Definition 38-68: *An APM Relation is an* ***APM Product Idealization Relation*** *if any of the primitive attributes in the list of related attributes* { pa_1 , pa_2 , … , pa_n } *is an APM Idealized Attribute.*

Analyzable Product Model (APM), Manufacturable Product Model (MPM), and Product Model (PM)

Essentially, an Analyzable Product Model is a container for the all APM Source Sets and APM Source Set Links that define an analysis-oriented view of a given product type. In other words, the APM provides a single point of entry to all the domains, attributes (product and

idealized) and relations that constitute an analyzable model of the product type. A given product type may have more than one APM: for example, a PWA may have an APM for thermomechanical analysis and another for cooling analysis. When an APM is instantiated with data, it is called and APM Instance (not to be confused with APM *Domain* Instance, which is an instance of an APM Domain – see Subsection 44).

With this in mind, an ***APM for a product type p*** is defined as follows:

$apm_i^{(p)} = (\ name\ ,\ \{\ s_1\ ,\ s_2\ ,\ \dots\ ,\ s_n\ \}\ ,\ \{\ l_1\ ,\ l_2\ ,\ \dots\ ,\ l_m\ \}\ ,\ constraint\_network\ )$(Definition 38-69)

Where:

$apm_i^{(p)}$: is a specific APM for product type ***p***;

name is the name of the APM;

$\{\ s_1\ ,\ s_2\ ,\ \dots\ ,\ s_n\ \}$ is the ordered list of source sets of the APM;

$\{\ l_1\ ,\ l_2\ ,\ \dots\ ,\ l_m\ \}$ is the ordered list of source set links of the APM;

constraint_network is the constraint network of the APM[25].

Properties:

$s_j \in \mathcal{S}$;

$l_k \in \mathcal{L}$;

constraint_network $\in \mathcal{CN}$ (see Definition 38-77).

Individual APMs are grouped to form the ***Set of APMs of a Product Type p***, $\mathcal{APM}^{(p)}$, as follows:

$\mathcal{APM}^{(p)} = \{\ apm_1^{(p)}\ ,\ apm_2^{(p)}\ ,\ \dots\ ,\ apm_n^{(p)}\ \}$ (Definition 38-70)

[25] The constraint network is not required for *defining* an APM since it can be derived dynamically from the relations defined in the source sets and the source set links. Rather, *constraint_network* is included in this definition as a placeholder for the constraint network once it is derived in memory.

As mentioned before and as the definition above implies, there may be more than one APM for a given product ***p***. The reason for this is that, for modularity and efficiency purposes, it may be preferable to have several specialized APMs for a given product rather than a single, monolithic APM. Each specialized $apm_i^{(p)}$ could support a different *suite* of analyses[26] typically for a common product-discipline combination. For example, two APMs could be defined for the analysis of an *airplane wing* ($apm_1^{(p)}$ and $apm_2^{(p)}$ where ***p*** represents the wing); the first to support a suite of *structural* analyses, and the second to support a suite of *aeroelasticity* analyses. The determination of the number of APMs that should be defined for a given product (***n*** in Definition 38-70) should be based on the amount of information required by each individual analysis and the amount of information that these analyses share. The objective is to keep APMs small and simple, while at the same time share as much information as possible.

At this point, it is useful to define two other types of product models that are closely related to the APM: the Product Model and the Manufacturable Product Model:

> Definition 38-71: *a* ***Manufacturable Product Model*** *of a product type p ($\mathcal{MPM}^{(p)}$) is a product model that contains only the minimum necessary information to manufacture the product.*
>
> Definition 38-72: *a* ***Product Model*** *of a product type p ($\mathcal{PM}^{(p)}$) is a superset of the $\mathcal{MPM}^{(p)}$ obtained by adding additional product attributes to the set of $\mathcal{MPM}^{(p)}$ attributes.*

Note that, by definition, a MPM contains only *essential product attributes* (Definition 38-62). A given product type p may have several alternative MPMs, depending on which attributes are considered to be essential for manufacturing.

The additional attributes defined in a PM are *redundant product attributes* (Definition 38-63). These added attributes are considered redundant since they are not required to manufacture the part and since most can be derived from MPM attributes. The product relations needed to derive these additional product attributes are also defined as part of the PM.

---

[26] A *suite* of analyses is a group of related analyses.

In light of these two new definitions, an APM $apm_i^{(p)}$ can be now viewed as the result of adding idealized information to a *subset* of a PM $\mathcal{PM}^{(p)}$. In other words:

$apm_i^{(p)}$ = Subset($\mathcal{PM}^{(p)}$) + idealized attributes + product idealization relations

Figure 38-32 shows the relationship between the PMs, MPMs and APMs.

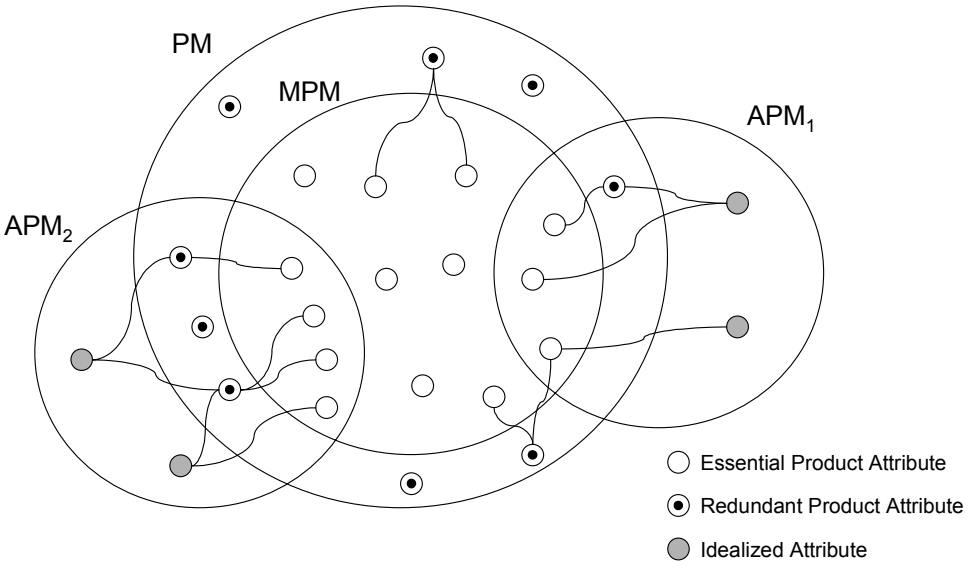


Figure 38-32: Relationship Between PMs, MPMs, and APMs

Next, when and APM is actually populated with instances, it is called an ***APM Instance***, defined as:

$apm_{ij}^{p} = (apm_i^{(p)}, \{ si_1, si_2, \ldots, si_n \})$ (Definition 38-73)

Where:

apm_{ij}^{p} : is a specific APM instance of APM $apm_i^{(p)}$;

$\{ si_1, si_2, \ldots, si_n \}$ is the ordered list of source set instances.

Finally, two more membership relationships will be defined. The first is between APM Source Sets and APMs:

$s_i \in^* apm_i^{(p)}$ iff $s_i \in \{ s_1 , s_2 , \dots , s_n \}$ from Definition 38-69 (Definition 38-74)

In other words, a source set s_i *belongs* to an APM $apm_i^{(p)}$ if it belongs to the list of source sets of $apm_i^{(p)}$.

The second membership relationship is defined between APM Source Set Links and APMs as follows:

$l_i \in^* apm_i^{(p)}$ iff $l_i \in \{ l_1 , l_2 , \dots , l_m \}$ from Definition 38-69 (Definition 38-75)

In other words, a source set link l_i *belongs* to an APM $apm_i^{(p)}$ if it belongs to the list of source sets links of $apm_i^{(p)}$.

Constraint Networks

A constraint network is an alternate way to represent APM Relations[27]. In a constraint network, variables[28] and relations are represented as vertices of a *simple graph* (a simple graph is one whose edges are undirected, that does not have multiple edges between the same two vertices, and that does not allow edges from a vertex to itself (Rosen 1995)). Each node in a constraint network may be either a *variable* or a *relation*. Variables can only be connected to relations and relations can only be connected to variables. This representation allows to determine what variables are affected by the change in value of a given variable, or what relations are required to calculate the value of a given variable. A graphical representation for constraint networks will be introduced in Subsection 57, and how APM Relations are mapped into constraint networks will be discussed in Subsection 59.

A ***Constraint Network*** is defined as follows:

---

[27] In this work, the terms *relation* and *constraint* are used interchangeably. They both refer to a mathematical expression that relates primitive attributes or variables.

[28] The term "variable" is used in constraint networks instead of "attributes" as in the APM, in part to highlight the fact that the concept of constraint networks is independent from the concept of APM. In other words, constraint networks may be applied in other contexts outside APMs.

$cn_i = (\{ cnr_1 , cnr_2 , \ldots , cnr_n \} , \{ cnv_1 , cnv_2 , \ldots , cnv_m \})$ (Definition 38-76)

Where:

cn_i: is a specific constraint network;

$\{ cnr_1 , cnr_2 , \ldots , cnr_n \}$ is a list of constraint network relations;

$\{ cnv_1 , cnv_2 , \ldots , cnv_m \}$ is a list of constraint network variables.

Properties:

$cnr_j \in CNR$, where CNR is defined in Definition 38-79 below;

$cnv_k \in CNV$, where CNV is defined in Definition 38-81 below.

Individual Constraint Networks are grouped to form the ***Set of Constraint Networks***, CN, as follows:

$CN = \{ cn_1 , cn_2 , \ldots , cn_n \}$ (Definition 38-77)

Constraint Network Relations are defined as follows:

$cnr_i = (name , expression , \{ cnv_1 , cnv_2 , \ldots , cnv_m \})$ (Definition 38-78)

Where:

cnr_i: is a specific Constraint Network Relation;

name is the name of the relation;

expression is the mathematical expression that relates the variables connected to this relation;

$\{ cnv_1 , cnv_2 , \ldots , cnv_m \}$ is a list of constraint network variables connected to this relation.

Properties:

$expression \in \mathbb{S}$;

$cnv_j \in CNV$.

Individual Constraint Network Relations are grouped to form the ***Set of Constraint Network Relations***, CNR, as follows:

$$CNR = \{ cnr_1 , cnr_2 , \ldots , cnr_n \} \quad \text{(Definition 38-79)}$$

A ***Constraint Network Variable***, on the other hand, is defined as follows:

$$cnv_i = (name , \{ cnr_1 , cnr_2 , \ldots . , cnr_n \}) \quad \text{(Definition 38-80)}$$

Where:

cnv_i: is a specific Constraint Network Variable;

name is the name of the variable;

$\{ cnr_1 , cnr_2 , \ldots . , cnr_n \}$ is a list of constraint network relations to which this variable is connected.

Properties:

$cnr_j \in CNR$.

Individual Constraint Network Variables are grouped to form the ***Set of Constraint Network Variables***, CNV, as follows:

$$CNV = \{ cnv_1 , cnv_2 , \ldots , cnv_n \} \quad \text{(Definition 38-81)}$$

A ***Constraint Network Node*** is either a Constraint Network Relation or a Constraint Network Variable:

$$cnn_i \in CNR \; i \; CNV \quad \text{(Definition 38-82)}$$

Finally, individual Constraint Network Nodes are grouped to form the ***Set of Constraint Network Nodes***, CNN, as follows:

$CNN = \{ cnn_1 , cnn_2 , \ldots , cnn_n \}$ (Definition 38-83)

APM Definition Languages

The APM Representation includes two definition languages. The first, called ***APM Structure Definition Language*** (APM-S), is used to define the source sets, domains, attributes, relations and source set links that make up the structure of an APM. The APM-S definition of an APM is stored in a file known as the ***APM Definition File***. The second language, called ***APM Instance Definition Language*** (APM-I), is used to define instances of the domains defined in the APM Definition File. As a language used to define instances of domains, APM-I serves the same purpose of any other instance definition language or format. An example of another format that can be used to define APM instances is STEP P21 (also used in the test cases of this work presented in Chapter 83).

APM Structure Definition Language

As the name suggests, the APM Structure Definition Language (APM-S) is used to define the structure of APMs. APM-S was styled after STEP's data definition language EXPRESS (ISO 10303-11 1994; Schenck and Wilson 1994), but adds semantics specific to the problem of design-analysis integration (such as idealized attributes, product idealization transformations, multi-level attributes, etc.).

APM-S definitions are stored in a file called the APM Definition File. The first task client applications must perform in order to use an APM is to parse these definitions, since they describe the structure of the data, contain instructions on how to link the data coming from different design repositories, and define the mathematical relationships needed to calculate derived or idealized attributes.

This subsection presents the context-free grammar (CFG)[29] that defines the syntax of the APM-S language. A CFG can then be re-written in terms of utility-specific definitions for lexical analyzer- and parser-generation utilities. The first type of utilities – the lexical

[29] A CFG specifies the tokens and productions (also known as rewriting rules) that define a compilable language (Fisher and J. 1988).

analyzer-generation utilities - takes a *lexical specification file* as input. This file defines the *tokens* or special words that define the language. The output of these utilities is a program (in the case of Lex, a C program) called *lexical analyzer* (also known as *lexer* or *scanner*) which can read a text file, identify the tokens and pass them to the parser for further processing. Examples of lexical analyzer-generation tools are Lex (Levine, Mason et al. 1995) and Jlex (Elliot 1997). As a reference, Appendix E includes the lexical specification for APM-S used in this work as the input to JLex.

The second type of utilities – the parser-generation utilities – takes a *grammar* or *parser specification file* as input. This file defines the *grammars* (series of rules used to recognize syntactically valid input) and the *actions* (code that is executed when a rule fires) of the parser. The output of these utilities is a program (in the case of Yacc, also a C program) called *parser*, which receives the tokens from the lexer, recognizes if any grammar rule fires, and if so runs the corresponding actions. Examples of parser-generation utilities are Yacc (Levine, Mason et al. 1995) and Java-CUP (Hudson 1998). As a reference, Appendix F includes the parser specification for APM-S used in this work as the input to Java-CUP. The actions are defined in pseudocode form in Appendix D.

The notation used in this subsection to define the CFG for the APM-S language is the following:

1. **UPPERCASE BOLD** is used to indicate APM-S keywords (or *tokens*);
2. *Italics* are used to indicate user-defined names;
3. Single quotes (` `) are used to indicate symbols that must be written literally;
4. Square brackets ([ ]) are used to indicate optional terms;
5. Curly brackets ({ }) are used to group several options of a required term;
6. A vertical line (|) is used to indicate mutually exclusive (OR) options; and
7. The · symbol is used to indicate "is defined as".

The APM definition for the flap link example that has been used throughout the chapter will be used to illustrate the syntax. This APM definition can be found in Appendix Z.1 and is reproduced here in Figure 38-33.

```
APM flap_link;

SOURCE_SET flap_link_geometric_model ROOT_DOMAIN flap_link;

DOMAIN flap_link;
  ESSENTIAL part_number : STRING;
  IDEALIZED effective_length : REAL;
  sleeve_1 : sleeve;
  sleeve_2 : sleeve;
  shaft : beam;
  rib_1 : rib;
  rib_2 : rib;
  ESSENTIAL material : STRING;
 PRODUCT_RELATIONS
  pr1 : "<rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2";
  pr2 : "<rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2";
 PRODUCT_IDEALIZATION_RELATIONS
  pir1 : "<effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> - <sleeve_1.radius> - <sleeve_2.radius>";
  pir2 : "<shaft.wf> == <sleeve_1.width>";
  pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> - <shaft.tf> )";
  pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness> - <sleeve_2.thickness>";
END_DOMAIN;

DOMAIN sleeve;
  ESSENTIAL width : REAL;
  ESSENTIAL thickness : REAL;
  ESSENTIAL radius : REAL;
  center : coordinates;
END_DOMAIN;

DOMAIN coordinates;
  ESSENTIAL x : REAL;
  ESSENTIAL y : REAL;
END_DOMAIN;

DOMAIN beam;
  critical_cross_section : MULTI_LEVEL cross_section;
  length : REAL;
  ESSENTIAL tf : REAL;
  ESSENTIAL tw : REAL;
  ESSENTIAL t2f : REAL;
  ESSENTIAL wf : REAL;
  ESSENTIAL hw : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir5 : "<critical_cross_section.detailed.tf> == <tf>";
  pir6 : "<critical_cross_section.detailed.tw> == <tw>";
  pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
  pir8 : "<critical_cross_section.detailed.wf> == <wf>";
  pir9 : "<critical_cross_section.detailed.hw> == <hw>";
END_DOMAIN;

MULTI_LEVEL_DOMAIN cross_section;
  detailed : detailed_I_section;
  simple : simple_I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir10 : "<detailed.wf> == <simple.wf>";
  pir11 : "<detailed.hw> == <simple.hw>";
  pir12 : "<detailed.tf> == <simple.tf>";
  pir13 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;

DOMAIN simple_I_section SUBTYPE_OF I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir14: "<area> == 2*<wf>*<tf> + <tw>*<hw>";
END_DOMAIN;

DOMAIN detailed_I_section SUBTYPE_OF I_section;
  IDEALIZED t1f : REAL;
  IDEALIZED t2f : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir15: "<area> == <wf>*( <t1f> + <t2f> ) + <tw>*( <t2f> - <t1f> ) + <tw>*<hw>";
  pir16: "<t1f> == <tf>";
END_DOMAIN;

DOMAIN I_section;
  IDEALIZED wf : REAL;
  IDEALIZED tf : REAL;
  IDEALIZED tw : REAL;
  IDEALIZED hw : REAL;
  IDEALIZED area : REAL;
END_DOMAIN;

DOMAIN rib;
  ESSENTIAL base : REAL;
  ESSENTIAL height : REAL;
  length : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET flap_link_material_properties ROOT_DOMAIN material;

DOMAIN material;
  ESSENTIAL name : STRING;
  stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN material_levels;
  temperature_independent_linear_elastic : linear_elastic_model;
  temperature_dependent_linear_elastic : temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;

DOMAIN linear_elastic_model;
  IDEALIZED youngs_modulus : REAL;
  IDEALIZED poissons_ratio : REAL;
  IDEALIZED cte : REAL;
END_DOMAIN;

DOMAIN temperature_dependent_linear_elastic_model;
  IDEALIZED transition_temperature : REAL;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
  flap_link_geometric_model.flap_link.material == flap_link_material_properties.material.name;
END_LINK_DEFINITIONS;

END_APM;
```

Figure 38-33: APM Definition for the Flap Link Example

The top construct defined in any APM Definition File is the APM. In order to define an APM, a name, a list of source sets, and an optional list of source set links must be specified. The syntax for defining APMs is the following:

> apm_definition · **APM** *apm_name* `**;**` source_sets [source_set_links] **END_APM** `**;**`

In Figure 38-33, the first and the last lines define the flap link APM:

```
APM flap_link;
   <source set definitions>
   <source set links definitions>
END_APM;
```

Sources set definitions require a name for the source set, a root domain name, and a set of domains. The syntax for defining source sets is the following:

source_set · **SOURCE_SET** *source_set_name* **ROOT_DOMAIN** *root_domain_name* `**;**` domains **END_SOURCE_SET** `**;**`

For example, the following lines define the two source sets of APM `flap_link`:

```
SOURCE_SET flap_link_geometric_model ROOT_DOMAIN flap_link;
   <domain definitions>
END_SOURCE_SET;

and

SOURCE_SET flap_link_material_data ROOT_DOMAIN material;
   <domain definitions>
END_SOURCE_SET;
```

Domain definitions are the heart of an APM-S definition. Two types of domains can be defined in an APM: regular domains and multi-level domains. The syntax for defining domains is the following:

domain ·

DOMAIN *domain_name* [**SUBTYPE_OF** *supertype_domain_name*] `**;**` [attributes] [relations] **END_DOMAIN** `**;**`

|

MULTI_LEVEL_DOMAIN *domain_name* `**;**` attributes [relations] **END_MULTI_LEVEL_DOMAIN** `**;**`

Regular domains are defined with a domain name, an optional list of attributes, and an optional list of relations. They may have one supertype domain, which must be declared as well. Multi-level domains are defined in a similar way; but they cannot have supertypes and their list of attributes - which corresponds to the list of levels in a multi-level domain - is required (it does not make sense to define a multi-level domain without levels).

An example of a definition of regular domain from Figure 38-33 is the definition of domain `flap_link`:

```
DOMAIN flap_link;
   <attributes>
```

```
   <relations>
END_DOMAIN;
```

And an example of a definition of a multi-level domain is the definition of multi-level domain `cross_section`:

```
MULTI_LEVEL_DOMAIN cross_section;
   <attributes>
   <relations>
END_MULTI_LEVEL_DOMAIN;
```

The syntax for defining attributes is the following:

attribute · [**ESSENTIAL** | **IDEALIZED**] *attribute_name* `**;**` [**LIST** `**[**` lower_bound `**,**` upper_bound `**]**` **OF**] { **REAL** | **STRING** | [**MULTI_LEVEL**] *domain_name* } `**;**`

lower_bound · { 0, 1, 2 , … }

upper_bound · { 1, 2, … | `**?**` } (must be greater than lower_bound)

By default, domain attributes are **PRODUCT** attributes (see subsection 47). Alternatively, they may also be declared as **ESSENTIAL** or **IDEALIZED**. The definition of an attribute requires an attribute name and a domain. The domain of an attribute can be a primitive domain (**REAL** or **STRING**), a complex domain (regular or multi-level), or an aggregate domain. In the case of aggregate domains, the lower and upper bounds of the aggregate must be specified as well.

For example, the following are the definitions for domains `flap_link` and `cross_section` from above, now including their attributes:

```
DOMAIN flap_link;
   part_number : STRING;
   IDEALIZED effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   material : STRING;
   <relations>
END_DOMAIN;
```

```
MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
   <relations>
END_MULTI_LEVEL_DOMAIN;
```

The definition of a domain can also include relations among its attributes. Relations can be **PRODUCT_IDEALIZATION_RELATIONS** or **PRODUCT_RELATIONS** (see Subsection 48). A domain may have one, both or none of these types of relations. The syntax for defining relations is:

relations · [**PRODUCT_IDEALIZATION_RELATIONS** productIdealizationRelations] [**PRODUCT_RELATIONS** productRelations]

Where:

productIdealizationRelation · *relation_name* `**:**` `**“**` *expression* `**”**` `**;**`

productRelation · *relation_name* `**:**` `**“**` *expression* `**”**` `**;**`

For example, domain `flap_link` has six product idealization relations (“pir1” through “pir6”):

```
DOMAIN flap_link;
   part_number : STRING;
   IDEALIZED effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   material : STRING;
 PRODUCT_IDEALIZATION_RELATIONS
   pir1 : "<effective_length> == <sleeve_2.center.x> -
           <sleeve_1.center.x> - <sleeve_1.radius> -
           <sleeve_2.radius>";
   pir2 : "<shaft.critical_cross_section.detailed.wf> ==
           <sleeve_1.width>";
   pir3 : "<shaft.critical_cross_section.detailed.hw> ==
           2*( <sleeve_1.radius> + <sleeve_1.thickness> )";
   pir4 : "<shaft.length> == <effective_length> -
<sleeve_1.thickness>
           - <sleeve_2.thickness>";
   pir5 : "<rib_1.length> == <sleeve_1.width>";
```

```
   pir6 : "<rib_2.length> == <sleeve_2.width>";
END_DOMAIN;
```

and multi-level domain `cross_section` has four product relations ("pr1" through "pr4"):

```
MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
 PRODUCT_RELATIONS
   pr1 : "<detailed.wf> == <simple.wf>";
   pr2 : "<detailed.hw> == <simple.hw>";
   pr3 : "<detailed.tf> == <simple.tf>";
   pr4 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;
```

The syntax for specifying relations indicates that an individual relation is defined by a relation name followed by a quoted string containing a mathematical expression. The angle brackets (<>) enclosing the attribute names shown in the examples above are not really part of the syntax. They are used in this work only for implementation reasons (to make it easier to identify the attribute names from the rest of the mathematical expression). Admittedly, the specification presented in this work remains vague as to what is considered a valid expression in a relation. The mathematical operations and symbols that can be used in an expression, for instance, are not specified. As it will be discussed in Chapter 110, a more detailed specification of this syntax is required. This syntax should be generic enough to allow the utilization of any constraint solver, while at the same time extensible to take advantage of the more powerful capabilities of some constraint solvers. For this work, the syntax for defining expressions will be limited – at least informally - to the standard algebraic operations (+, - , * , /) and the aggregate operations (SUM, MAX, MIN, AVG) defined in Subsection 48.

After the individual source sets have been defined, the last block of definitions corresponds to the source set links. The definition of each source set link requires the names of the two attributes used as keys for the link and a logical operator. The syntax for defining source set links is the following:

source_set_links · **LINK_DEFINITIONS** link_definitions
END_LINK_DEFINITIONS ;

link_definition · *key_variable_name_1* logical_operator *key_variable_name_2* **`;`**

logical_operator = { `==` | `!=` | `<` | `<=` | `>` | `>=` }

The flap link APM defines only one source set link:

```
LINK_DEFINITIONS
   flap_link_geometric_model.flap_link.material ==
      flap_link_material_data.material.name;
END_LINK_DEFINITIONS;
```

The source set link above specifies that instances of `flap_link` from source set `flap_link_geometric_model` must be linked to instances of `material` from source set `flap_link_material_data` when the value of attribute `material` of the instance of `flap_link` is equal to the value of attribute `name` of the instance of `material`

APM Instance Definition Language

APM-I is a language that was developed for this work to represent *instances* of APM domains in terms of the values of its attributes. It is similar in scope to the STEP P21 format (Appendix A). APM-I can be seen as just an alternative format for defining design data; unlike APM-S, APM-I is not essential for defining or using APMs.

Instances of a domain are defined in APM-I by listing the full names of the attributes of the domain followed by their values. Full attribute names are constructed by concatenating the names of complex attributes with the names of its attributes until a terminal (primitive) attributes is reached, separating each name with a dot. For example, the full attribute name of the x coordinate of the center of `sleeve_1` in the flap link example is `sleeve_1.center.x`.

The syntax of APM-I is the following:

data ::= **DATA** `;` instances **END_DATA** `;`

instance ::= **INSTANCE_OF** *domain_name* `;` values **END_INSTANCE** `;`

value ::= simple_value | aggregate_value

simple_value ::= *attribute_name* `:` { string_value | real_value | unknown_value } `;`

string_value ::= `**"**`*string*`**"**`

real_value ::= r, where r ∈ ℝ

unknown_value ::= `**?**`

aggregate_value ::= aggregate_element `**:**` { string_value | real_value | unknown_value } `**;**`

aggregate_element ::= aggregate_names [`**.**` aggregate_name]

aggregate_name ::= *string* `**[**` integer `**]**` | *string*

APM-I does not require that *all* attributes of a domain be listed. However, it may be desirable (for implementation reasons or just for expressiveness) to list all the attributes, including those that do not have value. In these cases a question mark (?) is used in place of the value to represent an "unknown" value.

The following APM-I definition defines one instance of flap link (more than one instances may be defined in the same APM-I definition):

```
DATA;

INSTANCE_OF flap_link;
   part_number : "FLAP-001";
   effective_length : 12.5;
   sleeve_1.width : 1.5;
   sleeve_1.thickness : 0.5;
   sleeve_1.radius : 0.5;
   sleeve_1.center.x : 0.0;
   sleeve_1.center.y : 0.0;
   sleeve_2.width : 2.0;
   sleeve_2.thickness : 0.6;
   sleeve_2.radius : 0.75;
   sleeve_2.center.x : ?;
   sleeve_2.center.y : 0.0;
   shaft.length : ?;
   shaft.critical_cross_section.detailed.wf : ?;
   shaft.critical_cross_section.detailed.tf : 0.1;
   shaft.critical_cross_section.detailed.tw : 0.1;
```

```
   shaft.critical_cross_section.detailed.hw : ?;
   shaft.critical_cross_section.detailed.area : ?;
   shaft.critical_cross_section.detailed.t1f : ?;
   shaft.critical_cross_section.detailed.t2f : 0.15;
   shaft.critical_cross_section.simple.wf : ?;
   shaft.critical_cross_section.simple.tf : ?;
   shaft.critical_cross_section.simple.tw : ?;
   shaft.critical_cross_section.simple.hw : ?;
   shaft.critical_cross_section.simple.area : ?;
   rib_1.base : 10.00;
   rib_1.height : 0.5;
   rib_1.length : ?;
   rib_2.base : 10.00;
   rib_2.height : 0.5;
   rib_2.length : ?;
   material : "aluminum";
END_INSTANCE;

END_DATA;
```

The following is also a valid APM-I definition (suppressing the unknown values from the definition above):

```
DATA;

INSTANCE_OF flap_link;
   part_number : "FLAP-001";
   effective_length : 12.5;
   sleeve_1.width : 1.5;
   sleeve_1.thickness : 0.5;
   sleeve_1.radius : 0.5;
   sleeve_1.center.x : 0.0;
   sleeve_1.center.y : 0.0;
   sleeve_2.width : 2.0;
   sleeve_2.thickness : 0.6;
   sleeve_2.radius : 0.75;
   sleeve_2.center.y : 0.0;
   shaft.critical_cross_section.detailed.tf : 0.1;
   shaft.critical_cross_section.detailed.tw : 0.1;
   shaft.critical_cross_section.detailed.t2f : 0.15;
   rib_1.base : 10.00;
   rib_1.height : 0.5;
   rib_2.base : 10.00;
   rib_2.height : 0.5;
   material : "aluminum";
END_INSTANCE;
```

```
END_DATA;
```

Instances of aggregate elements are defined by adding the index number to the attribute name. For example, the following is the APM-I definition of an instance of domain `pwa`, which has an attribute called layup, which is a list of layer instances, each having a material name and a thickness (this particular instance has three layers):

```
INSTANCE_OF pwa;
   part_number : "PWA-123" ;
   layup[0].material : "copper";
   layup[0].thickness : 0.2;
   layup[1].material : "FR4";
   layup[1].thickness : 0.5;
   layup[2].material : "copper";
   layup[2].thickness : 0.7;
   layup[3].material : "FR4";
   layup[3].thickness : 0.1;
END_INSTANCE;
```

APM Graphical Representations

The APM Representation specifies three graphical representations that can be used as visual tools for developing, communicating, or documenting APMs. These representations are also used throughout this work to illustrate the concepts being introduced and to illustrate the test cases in Chapter 83.

Each representation conveys a particular aspect of the APM better than the others do, and therefore the three should be used in a complementary fashion. They are fully derivable from the lexical (APM-S) definition of the APM. These representations are:

1. APM EXPRESS-G Diagrams;
2. APM Constraint Schematics Diagrams; and
3. APM Constraint Network Diagrams.

Of these three graphical representations, only the Constraint Network Diagrams are original to this work. The other two are extensions or adaptations of existing graphical representations.

APM EXPRESS-G Diagrams

As explained in Appendix A, EXPRESS-G is the graphical representation of EXPRESS lexical models. This subsection briefly introduces EXPRESS-G and explains how it is used to represent APM domains. EXPRESS-G is described in detail in the STEP standard (ISO 10303-11 1994) and in (Schenck and Wilson 1994).

EXPRESS-G diagrams are useful for representing entities, their attributes (part-of relations), subtypes and supertypes (is-a relations). Although EXPRESS and EXPRESS-G were originally conceived as the data definition languages for the STEP standard, they can be used as general-purpose data definition languages to define non-STEP data models as well.

EXPRESS and EXPRESS-G will be used extensively in Section 65 to present the implementation of the APM Information Model. They were also used in a few occasions Section 41 to illustrate some of the APM fundamental concepts.

EXPRESS-G diagrams can also be used to represent the domains and attributes defined in a specific APM Definition. It is possible to translate the APM-S domains used to define APMs into EXPRESS and, consequently, into EXPRESS-G. Before explaining how this is done, a brief introduction to EXPRESS-G will be provided first.

EXPRESS-G has three basic types of symbols: definition, relation, and composition symbols. Definition and relation symbols are used to define the contents and structure of an information model. Composition symbols are used to organize EXPRESS-G diagrams across several physical pages.

A definition symbol is a rectangle enclosing the name of the thing being defined. The type of the definition is denoted by the style of the box. Symbols are provided for EXPRESS simple types, defined types, entity types and schemas. As shown in Figure 38-34, the symbol for a simple type is a solid rectangle with a double vertical line at its right end. The name of the simple type (Binary, Boolean, Integer, Logical, Number, Real and String are the predefined simple types offered by the EXPRESS language) is enclosed within the box. A user-defined data type is represented with a dashed box enclosing the name of the type (Figure 38-35). An entity is represented with a solid rectangle enclosing the name of the entity (Figure 38-36). A schema is represented with a solid rectangle divided in half by a horizontal line (Figure

38-37). The name of the schema is written in the upper half of the rectangle. The lower half of the symbol is left empty.

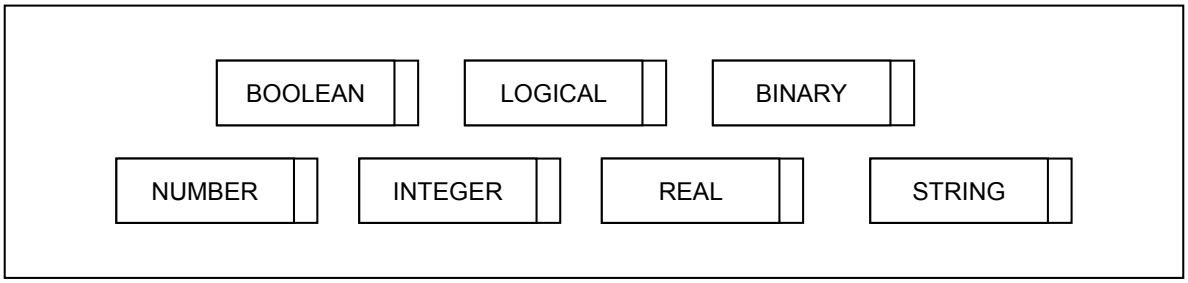


Figure 38-34: EXPRESS-G Simple Types Symbols

Figure 38-35: EXPRESS-G User-Defined Types Symbol

Figure 38-36: EXPRESS-G Entity Symbol

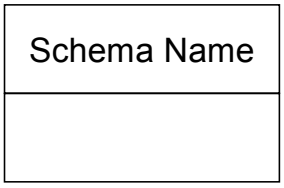


Figure 38-37: EXPRESS-G Schema Symbol

Relationship symbols are used to represent the relationship between two or more definition symbols. Relationship symbols are simply lines in three different styles: dashed, thick, and normal (Figure 38-38). Dashed lines are used to represent optional entity attributes. Thick lines are used to represent a subtype-supertype relationship between two entities. Normal solid lines are used to display all other relationships, the most important of which being the relationship between an entity and its non-optional attributes.

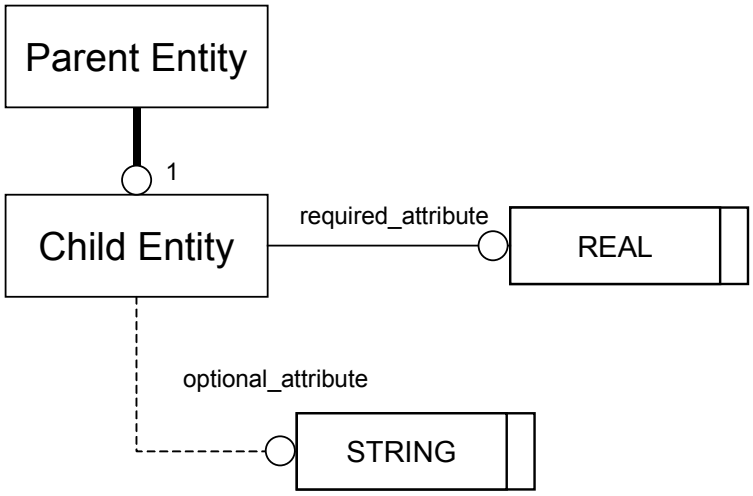


Figure 38-38: EXPRESS-G Relationship Symbols

Although in EXPRESS relationships are bi-directional, one of the directions is emphasized with an open circle on one end of the line that connects the two related entities. For example, if an entity `Man` has an attribute called `Wife` of type `Woman`, then the emphasized direction is from `Man` to `Woman` and the open circle is drawn on the `Woman` side of the relationship line.

The third and last type of symbols, composition symbols, is used to enable inter-page referencing in EXPRESS-G diagrams that span multiple pages. As illustrated in Figure 38-39, when two related entities `From_Definition` and `To_Definition` are separated into two different pages (pages 12 and 23), the relationship line between them is cut and terminated by an oval on each side. This oval contains referencing information to allow the reader to follow the relationship. On the `From_Definition` side, the information enclosed by the oval is of the form "`Page#`, `Reference#`, `Name`", where `Page#` is the number of the

page where the `To_Definition` is, `Reference#` is a number used to distinguish between potentially multiple references onto a page, and `Name` is the name of the referenced definition (in this case `To_Definition`). On the `To_Definition` side, the referencing information contained by the oval is of the form "`Page#`, `Reference#`, (`Page#1` , `Page#2` , …)", where `Page#` and `Reference#` are the same as in `From_Definition`'s oval, and (`Page#1` , `Page#2` , …) is a list of page numbers in which there are definitions that refer to `To_Definition`.

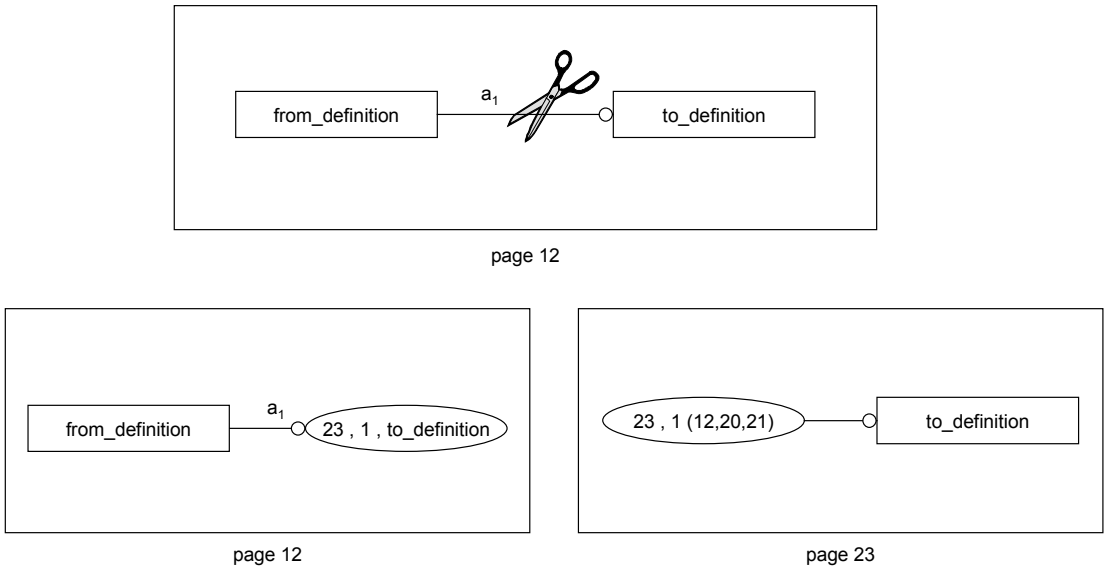


Figure 38-39: EXPRESS-G Composition Symbols

Since EXPRESS-G diagrams can be generated directly from EXPRESS, understanding how APM-S can be translated into EXPRESS is equivalent to understanding how APMs can be represented with EXPRESS-G. Since the APM-S language (introduced in Subsection 52) is very similar to EXPRESS, it is straightforward to translate APM-S into EXPRESS.

Each domain definition in APM-S becomes an entity in EXPRESS. The attributes of this domain become the attributes of its corresponding entity in EXPRESS. For example, the following APM-S definition:

```
DOMAIN flap_link;
   part_number : STRING;
```

```
   IDEALIZED effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   material : STRING;
END_DOMAIN;
```

is translated into EXPRESS as follows:

```
ENTITY flap_link;
   part_number : STRING;
   (* IDEALIZED *) effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   material : STRING;
END_ENTITY;
```

and its corresponding EXPRESS-G diagram is shown in Figure 38-40.

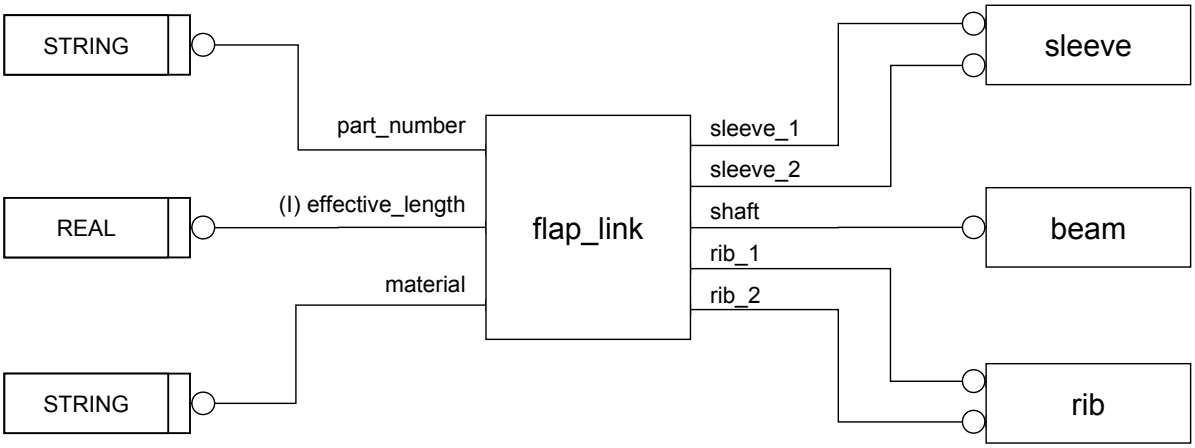


Figure 38-40: EXPRESS-G Diagram for Domain `flap_link`

Notice how the fact that attribute `effective_length` is an idealized attribute is lost during the translation. To remedy this (at least partially) the comment `(*IDEALIZED*)` is added

before the name of the attribute, as done above. In the EXPRESS-G diagram, the label "`(I)`" is added in front of the attribute name.

The concept of multi-level domains is not supported in EXPRESS. Therefore, APM-S Multi-Level domains are translated as if they were regular domains. This loss of information can be partially circumvented by adding the comment `(*MULTI_LEVEL*)` before the word `ENTITY`. There is not a corresponding symbol in EXPRESS-G for multi-level domains either, so a new symbol was added in this work to represent the concept of multi-level entities (even if they are not formally defined in EXPRESS). The symbol for multi-level entities is simply a box with a diagonal line in its upper-left corner.

For example, the following multi-level domain:

```
MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
END_MULTI_LEVEL_DOMAIN;
```

is translated into EXPRESS as a regular entity:

```
(* MULTI_LEVEL *)ENTITY cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
END_ENTITY;
```

and its corresponding EXPRESS-G diagram is shown in Figure 38-41.

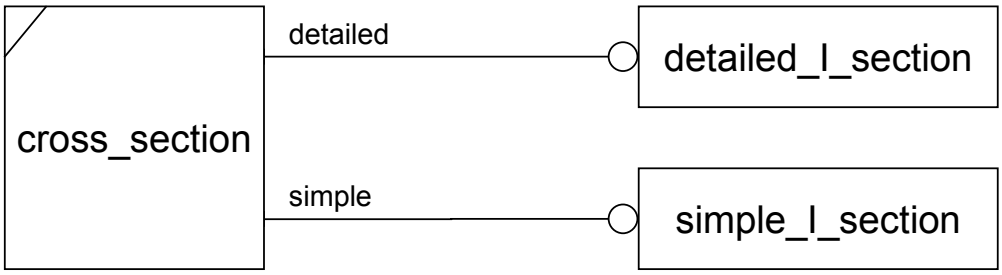


Figure 38-41: EXPRESS-G Diagram for Multi-Level Domain `cross_section`

APM Constraint Schematics Diagrams

Constraint schematics diagrams are a convenient way to represent domain part-of hierarchies and the relations between their attributes. Constraint schematics diagrams are based on directed graphs diagrams and were first used in (Peak 1993) to represent Analysis Building Blocks and Product Model-Based Analysis Models. Some additional symbols were added in this work in order to convey more APM-Specific concepts. The notation used in constraint schematics diagrams is summarized in Figure 38-42 and reproduced in Appendix I for future reference.

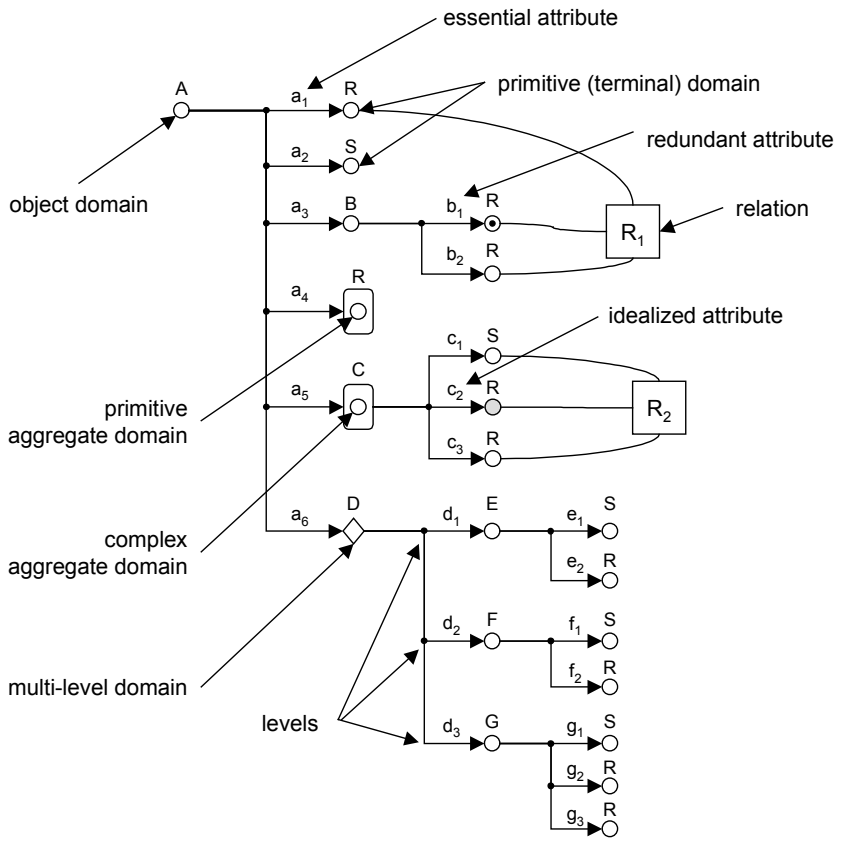


Figure 38-42: Basic Constraints Schematics Diagrams Notation

In constraint schematics diagrams, domains are represented with circles. The name of the domain is indicated in uppercase letters directly above the circle (for example, **A** through **G**

are domains in the figure). Domain attributes are represented with arrows coming out of the circle representing the domain, and are labeled with the name of the attribute in lowercase letters (e.g., $\mathbf{a_1}$, through $\mathbf{a_6}$, $\mathbf{b_1}$, $\mathbf{b_2}$, etc.). When the domain has more than one attribute, only one line is drawn coming out of the circle and small filled dots are used to branch off to the individual attributes. Attributes pointing to a shaded circle represent idealized attributes (for example, attribute $\mathbf{A.a_5.c_2}$ is an idealized attribute). Attributes pointing to a circle with a dot in the middle represent redundant attributes (for example, attribute $\mathbf{A.a_3.b_1}$ is a redundant attribute). The rest of the attributes (those pointing to regular circles) represent essential attributes (see Subsection 47)[30]. Attributes of a domain - and any relations in which they are involved - can be collapsed into their containing domain, providing an "encapsulated" alternative view of the objects and the relations among them. Hence, a relation may be potentially drawn between two or more complex domains, indicating that some of their internal attributes are related.

For clarity, attribute lines should only be drawn either horizontally or vertically. Each branch of the graph must end in primitive (terminal) domains, labeled **R** (for Real attributes) or **S** (for String attributes).

A circle enclosed by a rectangle represents an aggregate domain (attributes $\mathbf{A.a_4}$ and $\mathbf{A.a_5}$ in the figure point to aggregate domains). Aggregates are like domains but have *elements* instead of attributes. When the circle enclosed by the rectangle represents a primitive domain (that is, it is labeled **R** or **S**), the symbol indicates a primitive aggregate domain (attribute $\mathbf{A.a_4}$ points to a primitive aggregate). When the circle corresponds to a complex domain (that is, one that is not primitive, in other words, that has attributes) then it indicates a complex aggregate domain (attribute $\mathbf{A.a_5}$ points to a complex aggregate).

Diamond symbols are used to indicate multi-level domains. Multi-level domains have *levels* instead of attributes. Levels are represented in the same way as attributes in complex domains.

Labeled boxes indicate relations between attributes. The label inside the box indicates the name of the relation. Free-form lines are used to link the related attributes to the relation. Notice that, by definition of an APMRelation (see Definition 38-65), only attributes with

[30] Even though the circle represents the *domain* of the attribute, the category of the *attribute* is represented in the circle because it is easier to draw and reduces cluttering.

primitive domains (more specifically, attributes with *real* domains) can be connected to relations. In the example, $\mathbf{R_1}$ and $\mathbf{R_2}$ represent relations. $\mathbf{R_1}$ relates attributes $\mathbf{A.a_1}$, $\mathbf{A.a_3.b_1}$ and $\mathbf{A.a_3.b_2}$, and $\mathbf{R_2}$ relates attributes $\mathbf{A.a_5.c_1}$, $\mathbf{a_5.c_2}$ and $\mathbf{A.a_5.c_3}$.

Constraint Schematics do not show inheritance (is-a) hierarchies among domains. Attributes inherited from parent domains have to be repeated in the child domain every time it appears in the diagram. EXPRESS-G diagrams, introduced in the previous subsection, are more appropriate to show domain (is-a) hierarchies.

As shown in Figure 38-43, constraint schematics diagrams can also be used to represent domain *instances*. For this purpose, the labels indicating the names of primitive domains (**R** and **S**) are replaced with values. For example, attribute $\mathbf{A.a_1}$ in Figure 38-43 has a real value of 2.5 and attribute $\mathbf{A.a_2}$ has a string value of "string1". Instances of aggregate elements are represented by adding a sequential index enclosed by square brackets to the name of the attribute. For example, attribute $\mathbf{A.a_4}$ points to a list of reals, whose elements are $\mathbf{A.a_4[0]}$ (with a value of 7.8), $\mathbf{A.a_4[1]}$ (with a value of 11.11), and $\mathbf{A.a_4[2]}$ (with a value of 9.14). This indexing is also used when the elements of the aggregate are complex instances (such as in attribute $\mathbf{A.a_5}$, a list whose elements are of type **C**), but in this case each indexed element points to another object which, in turn, also has attributes. For example, attribute $\mathbf{A.a_5[0]}$ in the figure points to an instance of **C** (indicated by a circle labeled **C**) which, in turn, has three primitive attributes: $\mathbf{c_1}$ (with a value of "string2"), $\mathbf{c_2}$ (with a value of 7.31) and $\mathbf{c_3}$ (with a value of 8.09).

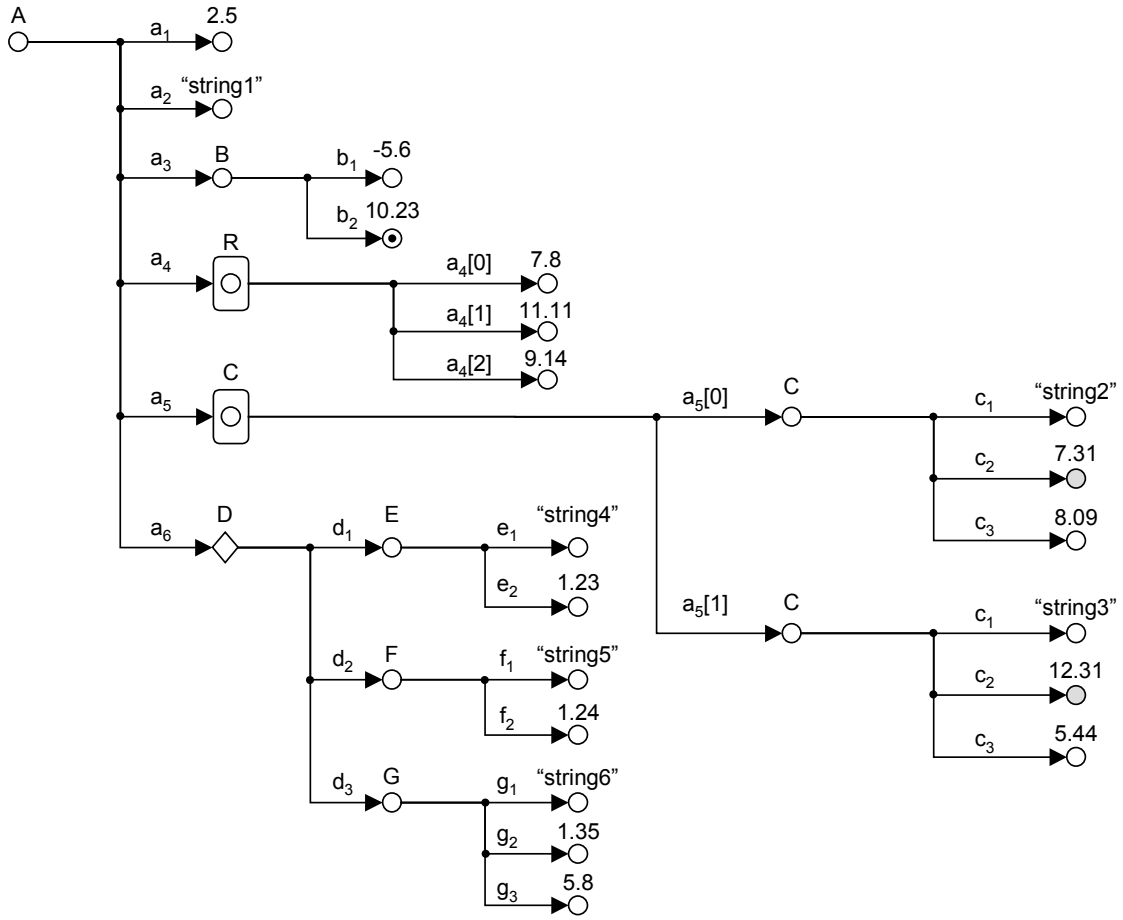


Figure 38-43: Representing Domain Instances with Constraints Schematics Diagrams

As an additional example of a constraint schematics diagram, Figure 38-9 shows the constraint schematics diagram for the flap link example used throughout this chapter. The APM Definition File from which this diagram was constructed can be found in Figure 38-7 or in Appendix Z.1.

APM Constraint Network Diagrams

APM Constraint Network Diagrams are used to represent constraint networks, which show how attributes are interconnected via relations. They are, in essence, a "flattened" view of the constraints schematics diagrams, where graphical part-of relations have been dropped making it easier to appreciate how APM attributes and relations are interconnected.

Constraint diagrams are useful, for example, to visualize which attributes are affected by the change of value of another attribute, or to determine which relations should be used to build the system of equations to solve for the value of a given attribute. They are also useful in figuring out possible input/output combinations for the values of the attributes (see Subsection 81).

As shown Figure 38-44, the notation for constraint network diagrams is very simple: relations are represented with boxes (labeled with the name of the relation) and attributes with circles. Attributes are connected to relations by curved lines and are labeled with their *full attribute names* (the full attribute name is obtained by concatenating the attribute names all the way up to the root of the domain hierarchy tree). Figure 38-44 also shows the constraint schematics diagram from which the constraint network diagram of this example is derived.

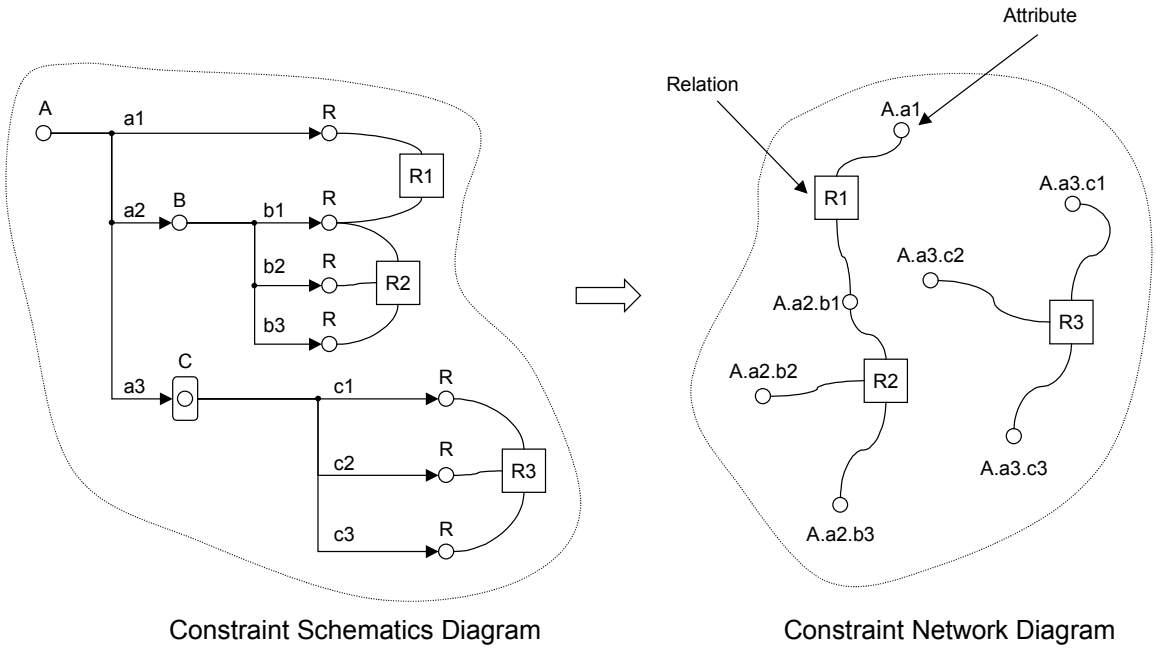


Figure 38-44: Constraint Network Diagrams

One practical caveat about constraint network diagrams: they can mislead one into thinking that attributes in one disconnected subgraph are independent of attributes in another

subgraph. However, this is not necessarily true: two attributes that belong to the same domain may be in separate subgraphs but their part-of relations to their common domain make them dependent on each other. A simple example is the Young's Modulus and the Poisson's ratio of a material: there may not be a relation between the two defined in the APM, but that does not mean that they are totally independent from each other.

APM Protocol

The APM Information Model presented in Section 41 defines the *data structure* of the APM Representation. In order to be of any value, APM data conforming to this structure must be easily accessible by APM client applications that need it to perform specialized tasks such as engineering analysis. In response to this, this section describes a minimal set of conceptual operations on APM constructs - collectively known as the ***APM Protocol*** - to allow access to and manipulation of APM-defined data.

The underlying assumption of this section is that these operations will be implemented in a computer environment (although as conceptual operations, they can be performed manually or in paper). By specifying the operations that must be supported by specific implementations of the APM Representation, the APM Protocol can be used to define programming interfaces to APM-defined data. Implementations of the APM Protocol in specific programming languages may be delivered as libraries of APM classes that application developers can then use to develop their APM-driven applications.[31] A prototype implementation of the APM Protocol developed by the author using the Java programming language will be presented in Chapter 64, and its utilization demonstrated with several real-world test cases in Chapter 83.

This section provides a high-level descriptive specification of *what* the operations are as opposed to specifying *how* they do it. Specific implementations of the APM Protocol will implement these operations in the way that is most convenient and efficient for the particular language being used. The next chapter will discuss how these operations were

[31] Although this specification is programming-language independent, it relies on object-oriented concepts and therefore is easier to implement using object-oriented programming languages. However, there are programming techniques that can be used to map object-oriented concepts into non-object-oriented languages. These techniques are discussed in (Rumbaugh, Blaha et al. 1991).

implemented in the prototype APM implementation developed by the author, as well as the specific algorithms used.

Although applications that operate on APM data through the APM operations may vary widely in purpose and complexity, a typical sequence of the high-level tasks performed by any APM-driven application using the APM Protocol is (Figure 38-45):

1. Load the APM definitions: includes key conceptual operations such as link the source set definitions and create the constraint network;
2. Load the source set data: includes linking the source set data;
3. Use the APM data; includes solving the constraint network; and
4. Save changes: includes unlinking the source set data.

As indicated in the figure, each of the four tasks listed above involves more than one APM operation. The subsections that follow explain each of these tasks in more detail.

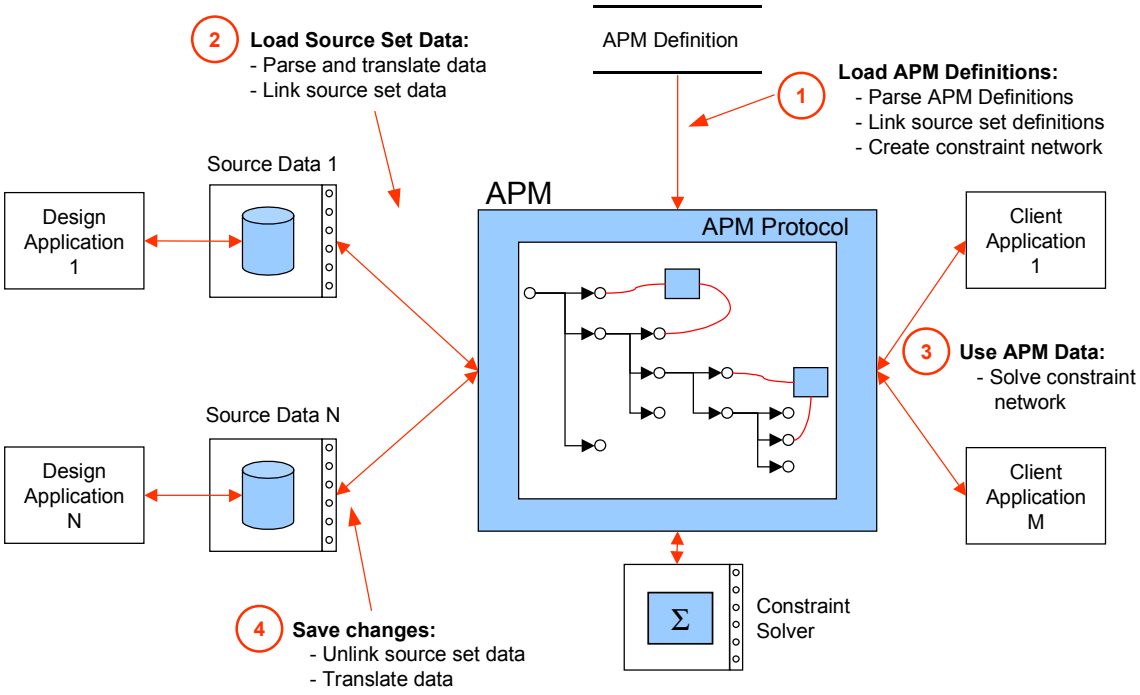


Figure 38-45: Typical High-Level Tasks Performed by APM-Driven Applications

APM Definition Loading

The APM Protocol must provide operations to parse APM definitions - described in APM-S and stored in the APM Definition File - and load it into memory to be accessed by client applications. Since APM definitions define both the structure of the APM data and the instructions on how to combine the design data coming from different sources, this operation must be performed by APM client applications prior to loading any design data into the APM.

The overall task of loading the APM definitions is illustrated in Figure 38-46. The figure shows the three main operations of this task: 1) parse and load the APM source set definitions, 2) link the APM source set definitions, and 3) create the constraint network. Upon completion of this task, the following information about the structure of the APM should be available in memory to be used by APM client applications:

1. The APM Source Sets, APM Domains, APM Attributes, APM Relations, and APM Source Set Links defined in the APM Definition File.
2. A linked version of the APM Domains, obtained by linking the domains of the individual source sets according to what is specified by the APM Source Set Links.
3. A constraint network representing the relations between APM Primitive Attributes in the APM.

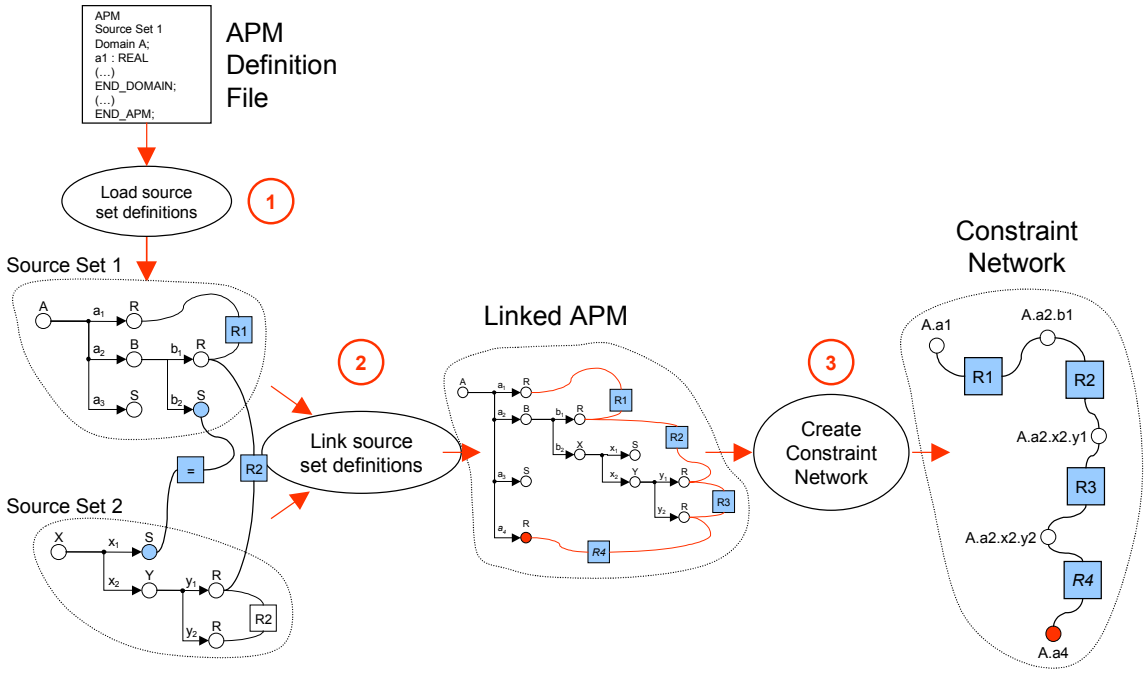


Figure 38-46: APM Definitions Loading Task

As illustrated in Figure 38-47, the first operation of this task - loading the APM source set definitions - should parse the APM Definition File that contains the APM-S definition of the APM and create the corresponding source set structures in memory. At this point, APM client applications must be able to query information about the structure (domains, domain hierarchy, attributes, relations) of the individual source sets.

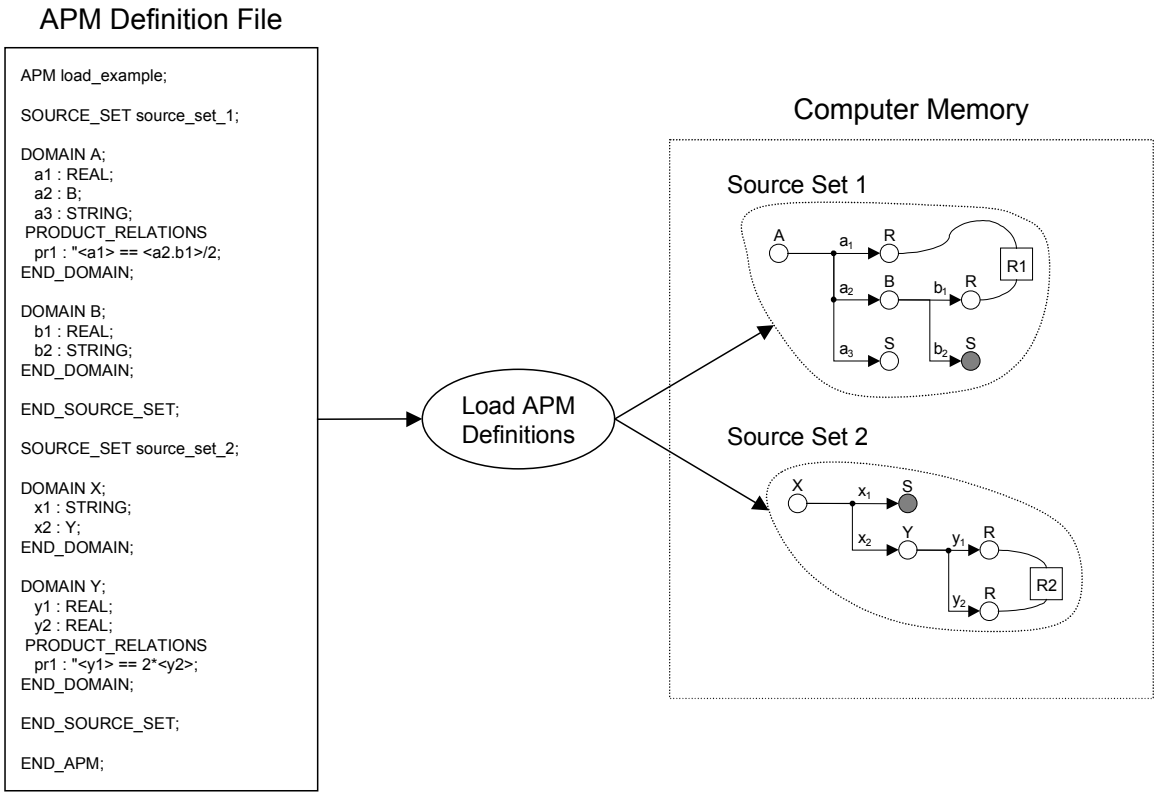


Figure 38-47: APM Definitions Loading Operation

After loading the APM definitions, the second operation of this task is to link these APM source set definitions according to what is specified by the APM Source Set Links defined in the APM. In order to illustrate how APM source set definitions are linked, consider the APM "`link_example`" defined in the APM Definition shown in Figure 38-48, and its corresponding constraint schematics representation in Figure 38-49 (a more formal explanation of how APM Source Set Links should be interpreted was provided in Subsection 46).

```
APM link_example;

SOURCE_SET setOne ROOT_DOMAIN A;

DOMAIN A;
  a1 : STRING;
  a2 : B;
  a3 : LIST[0,?] OF C;
END_DOMAIN;

DOMAIN B;
  b1 : REAL;
  b2 : STRING;
END_DOMAIN;

DOMAIN C;
  c1 : REAL;
  c2 : STRING;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET setTwo ROOT_DOMAIN X;

DOMAIN X;
  x1 : STRING;
  x2 : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET setThree ROOT_DOMAIN Y;

DOMAIN Y;
  y1 : STRING;
  y2 : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET setFour ROOT_DOMAIN Z;

DOMAIN Z;
  z1 : STRING;
  z2 : REAL;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
 setOne.A.a1 == setTwo.X.x1;
 setOne.A.a2.b2 == setThree.Y.y1;
 setOne.A.a3.c2 == setFour.Z.z1;
END_LINK_DEFINITIONS;

END_APM;
```

Figure 38-48: Source Set Link Example: APM Definition File

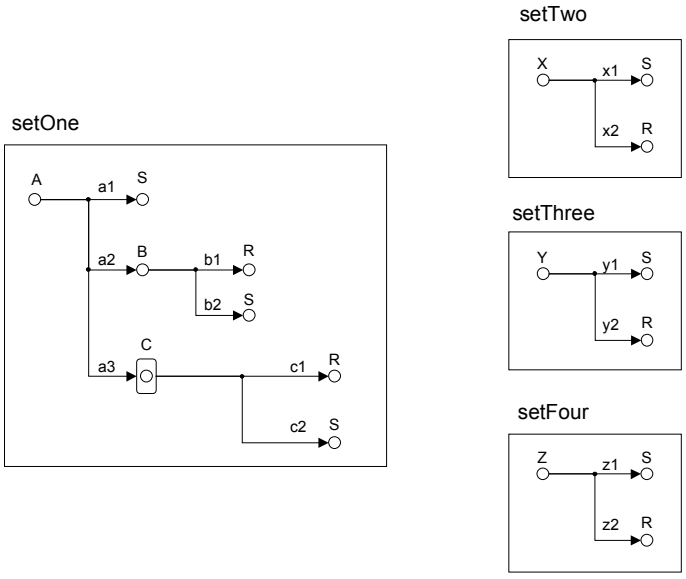


Figure 38-49: Source Set Link Example: Constraint Schematics Diagram

The APM defined in this file has four source sets (**setOne**, **setTwo**, **setThree**, and **setFour**) and three source set links (defined between the keywords **LINK_DEFINITIONS** and **END_LINK_DEFINITIONS** in the APM definition file shown of Figure 38-48). As illustrated in Figure 38-50, the first source set link links attribute **a1** of domain **A** in **setOne** with attribute **x1** of domain **X** in **setTwo,** the second links attribute **b2** of attribute **a2** of domain **A** in **setOne** with attribute **y1** of domain **Y** in **setThree,** and the third links attribute **c2** of attribute **a3** (which is an aggregate of **C**'s) of domain **A** in **setOne** with attribute **z1** of domain **Z** in **setFour**. The resulting linked APM is shown in Figure 38-51.

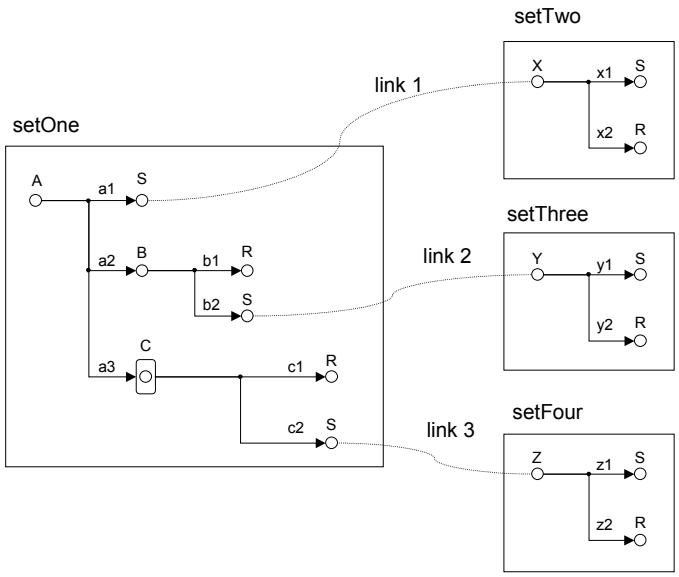


Figure 38-50:Source Set Link Example: Links

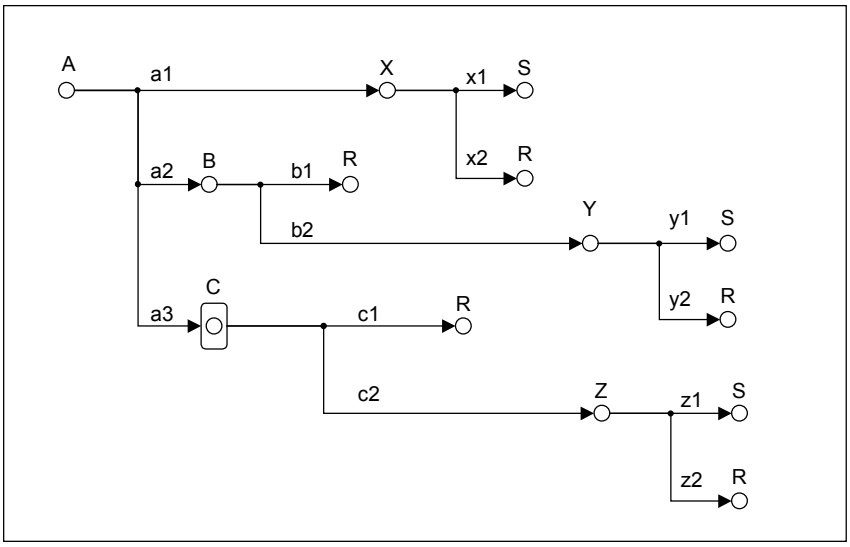


Figure 38-51: Source Set Link Example: Resulting Linked APM

After linking the APM definitions, APM client applications must be able to query information about the structure of the *linked* APM.

The third and last operation of this task is to create the constraint network of the APM (constraint networks and their graphical representation are described in Subsections 50 and 57, respectively). As it will be discussed in more detail in Subsection 81, the constraint network is used by the constraint-solving strategy to build the system of equations that will be used to resolve for the unknown value of a requested attribute. There is one constraint network for each APM, and it is built directly from the APM Relations contained in its APM Complex Domains. As illustrated in Figure 38-52, the constraint network is obtained by building a simple graph in which the nodes are the APM Relations and the APM Primitive Attributes connected to them.

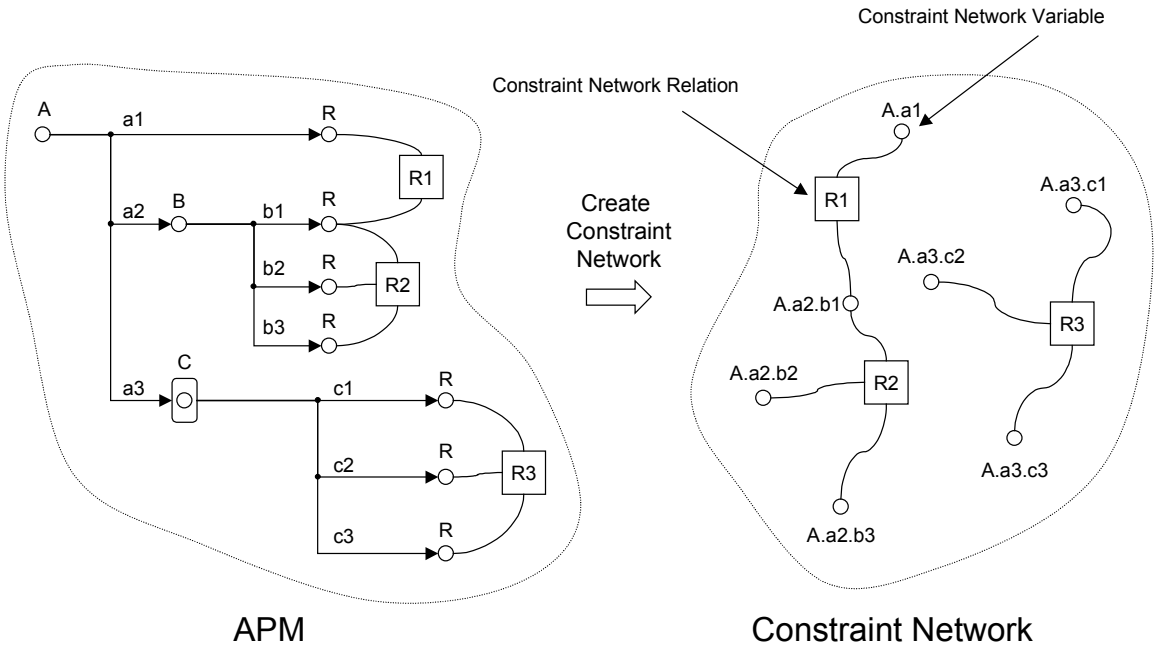


Figure 38-52: Constraint Network Creation Example

Source Set Data Loading

Normally once the APM definitions have been loaded and linked, the next task is to populate the APM structure with data coming from the design sources. This task consists of two main operations: load the source set data and link source set data.

The first main operation – load the source set data - is illustrated in Figure 38-53. This operation loads instances of the domains defined in the APM from various design repositories. As discussed in Subsection 45, APM source sets are purposely defined so that instances of domains in the same source set come from the same design repository. For example, in Figure 38-53 instances of domain **A** (defined in Source Set 1) come from the first repository ("Design Repository 1"), and instances of domain **X** (defined in Source Set 2) come from the second repository ("Design Repository 2").

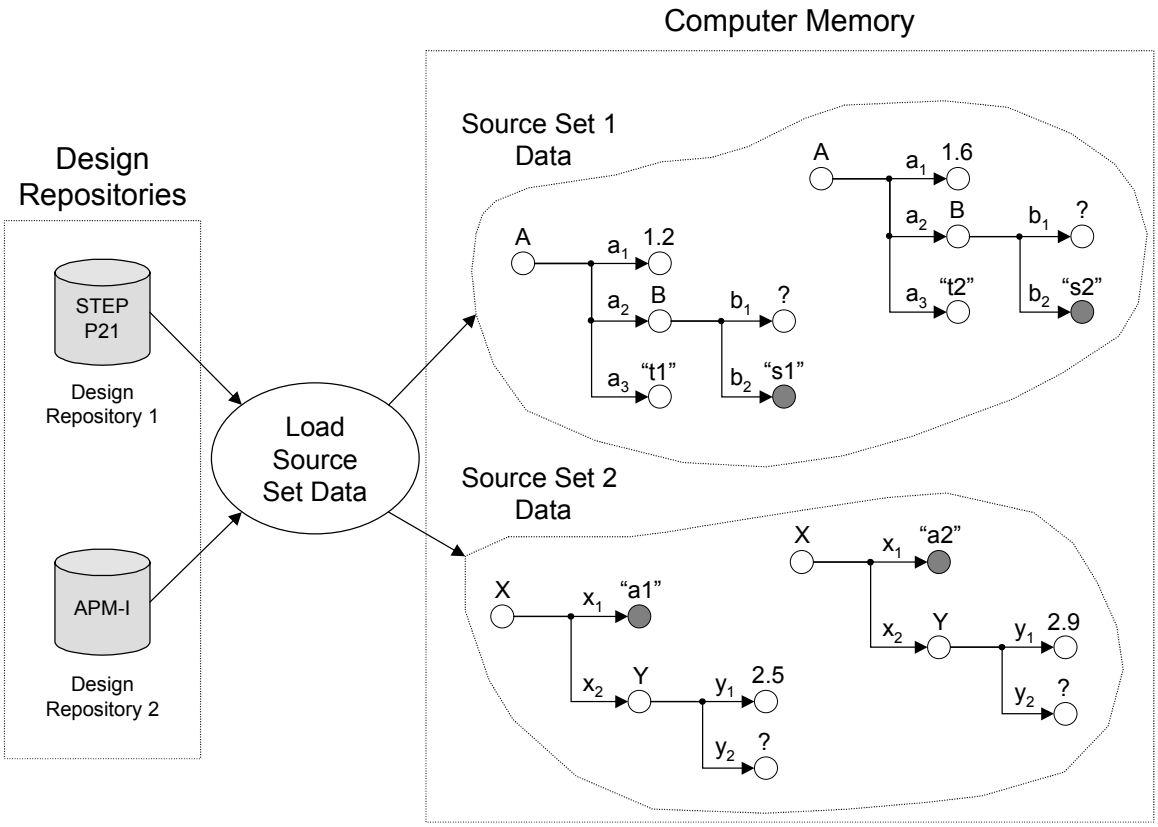


Figure 38-53: Load Source Set Data Operation

As also illustrated in Figure 38-53, data in the design repositories may be stored in a variety of formats. In this figure, for example, design data is stored in two different formats: STEP P21 (Appendix A) and APM-I (Subsection 53). It would be impractical to design a load source set data operation that can read all the formats in which the design data may be possibly stored. Instead, a *source data wrapping approach* is proposed, which relieves the load source set data operation from having to interpret the specific formats of the design data.

With this approach, special objects called "source set data wrapper objects" provide the necessary translation services, isolating the load source set data operation from the formatting details of the specific design repositories. These wrapper objects parse the design data and transform it into a neutral form understood by the load source set data operation. Hence, the load source set data operation only needs to handle one format, since different wrapper objects handle the specific format conversions that may arise. The main idea of the

source data wrapping approach is to shift the burden of the formatting details from the load source set data operation to the wrapper objects and keep the communication between the two as simple as possible.

Figure 38-54 illustrates the utilization of these wrapper objects in the example of Figure 38-53. Here, the "STEP wrapper object" parses the STEP P21 data of "Design Repository 1", and the "APM-I wrapper object" parses the APM-I data of "Design Repository 2".

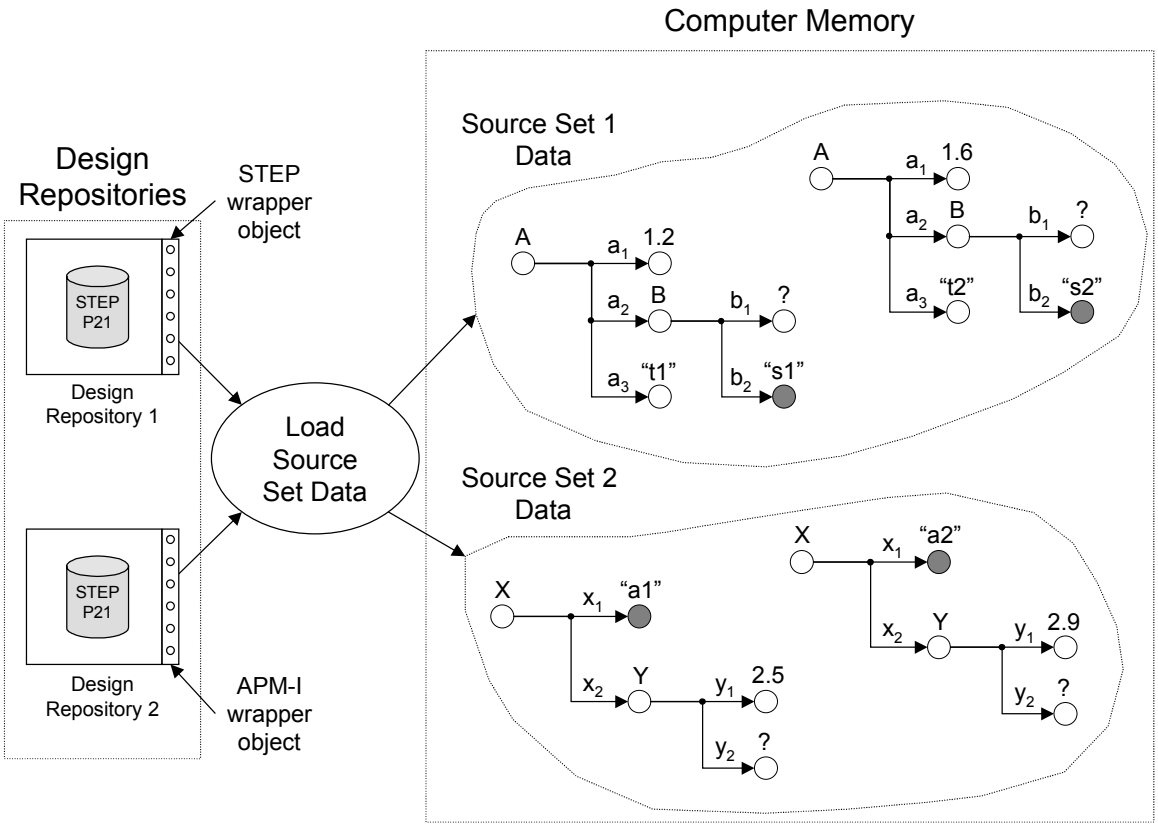


Figure 38-54: Source Data Wrapping Approach

This wrapping technique is based on an important assumption that requires some examination. When the load source data operation requests the source data wrapper object to get the instances of a certain domain from the design repositories, the domain must in fact *exist* in the design representation. Moreover, the *structure* of this domain (that is, its attributes and their types) must match the structure of the corresponding source set in the

APM. This is illustrated in Figure 38-55, which shows how data about the geometry of a plate with a hole - created by a solid modeler - is loaded into the APM. The APM in this example (labeled "APM's Representation" in the figure) defines a source set called **`plate_geometry`**, which includes a domain called **`plate`** with attributes **`length`**, **`height`**, **`thickness`**, **`hole`**, **`d`**, and **`material`**. Essentially, this is how the APM "expects" the data to come from the source data wrapper. In this figure, the solid modeler creates the data in a format that matches this "expected" representation (labeled "Solid Modeler's Representation" in the figure). Hence, the source data wrapper only has to perform a *syntactic* translation of the data, to convert the STEP instances (in this example) created by the solid modeler to corresponding **`APMDomainInstances`**.

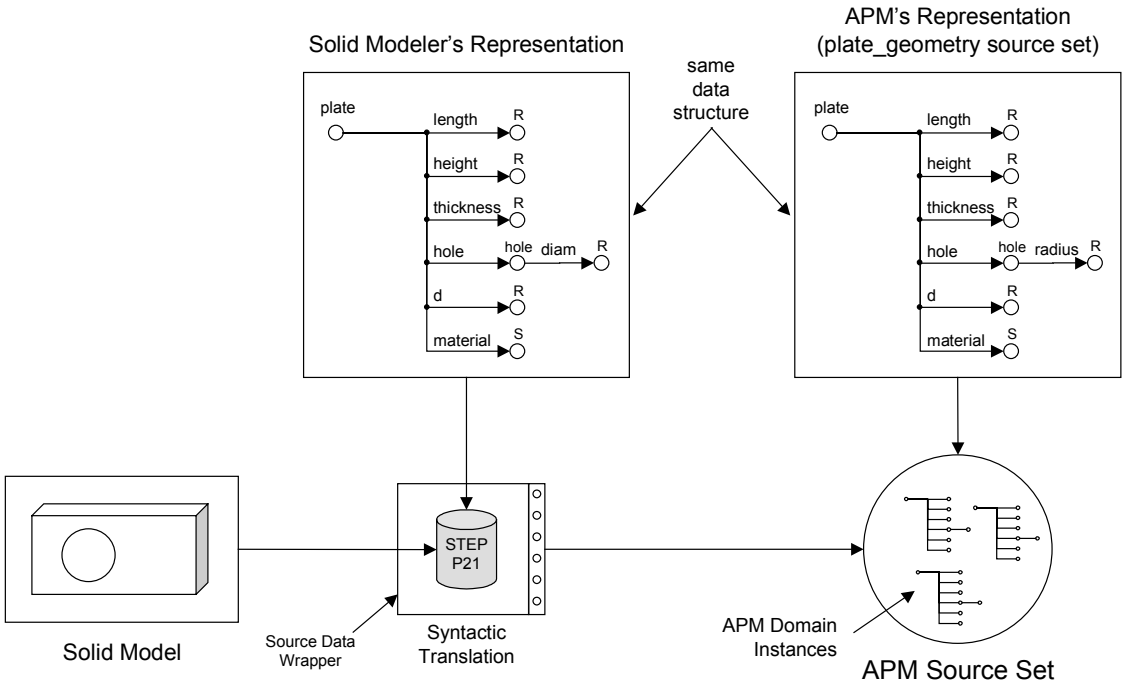


Figure 38-55: Loading Design Data into the APM (Requiring Syntactic Translation Only)

However, the situation illustrated in Figure 38-55 above is not quite realistic. In general, as described in Chapter 1, there is a semantic mismatch between design analysis that must be dealt with at some point. The structure of the data coming from the design applications might, in fact, differ significantly from what the APM expects. This more realistic situation is

illustrated in Figure 38-56, which shows the same example of the plate now considering the semantic mismatch between the representations of the solid modeler and the APM. In this example, the solid modeler represents the plate as a solid resulting from subtracting a "**cylinder**" from a "**block**", whereas the APM represents it as a "**plate**" with a "**hole**". Hence, although these two representations describe the same thing (a plate with a hole), they do it in very different ways.

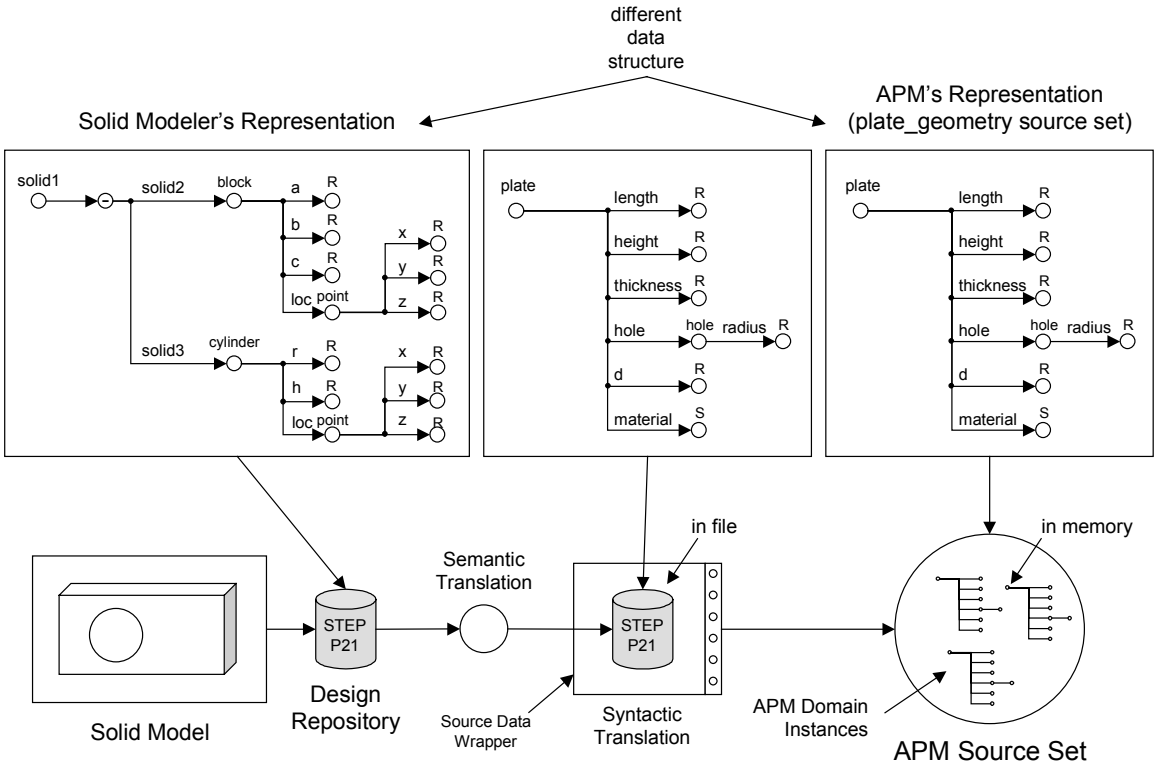


Figure 38-56: Loading Design Data into the APM (Requiring Semantic and Syntactic Translation)

The approach adopted by the APM technique to deal with this semantic mismatch between design and analysis is to perform a two-stage translation. In the first stage, as shown in Figure 38-56, a *semantic* translation takes place. The purpose if this semantic translation is to transform the design data - by rearranging its structure, simplifying it, and changing attribute names - so that it matches the data structure specified by the APM. The mechanism in which the instances are physically stored (in the example of the figure, STEP P21 files) can still be

the same. Next, source data wrappers perform a *syntactic* translation, transforming the data from whatever form resulted from the semantic translation (P21 in this example) into APM Domain Instances in computer memory. As shown in Figure 38-57, this semantic-syntactic translation process is repeated for each source set before linking them into a single, unified APM.

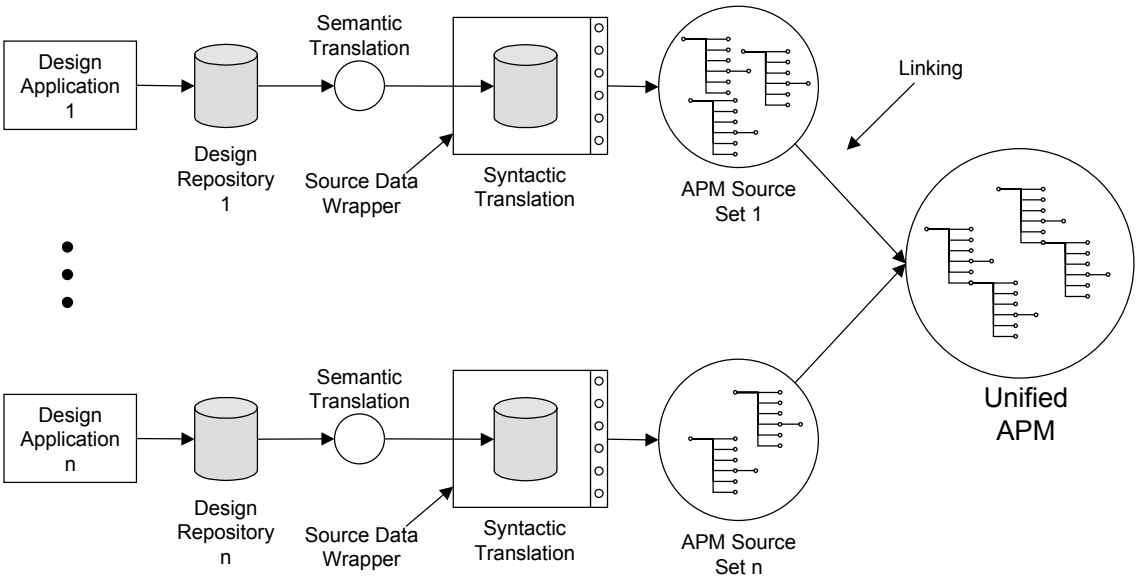


Figure 38-57: Semantic and Syntactic Translation for Each Source Set

There are several reasons for adopting the approach of resolving the semantic mismatch before the data is loaded into the APM. First, by doing so, the structure of the APM does not become unnecessarily complex, because it does not have to match the structure of the design representation (which often contains more detail than needed for analysis) in order to be able to load the data. Second, as a consequence of the first reason, the relations required to idealize product information in the APM become less complex. And third, it makes the APM independent from the design representations of the specific design tools. Therefore, if a design representation is replaced with another, the structure of the APM remains unaffected.

The specific mechanism used to perform this semantic translation depends, of course, on the representations involved. For example, as shown in Figure 38-58, if the solid modeler of

the example above stores its data in STEP AP203 format, a STEP mapping language such as EXPRESS-X (Spooner, Hardwick et al. 1996) could be used to map AP203 data to the APM Information Model.

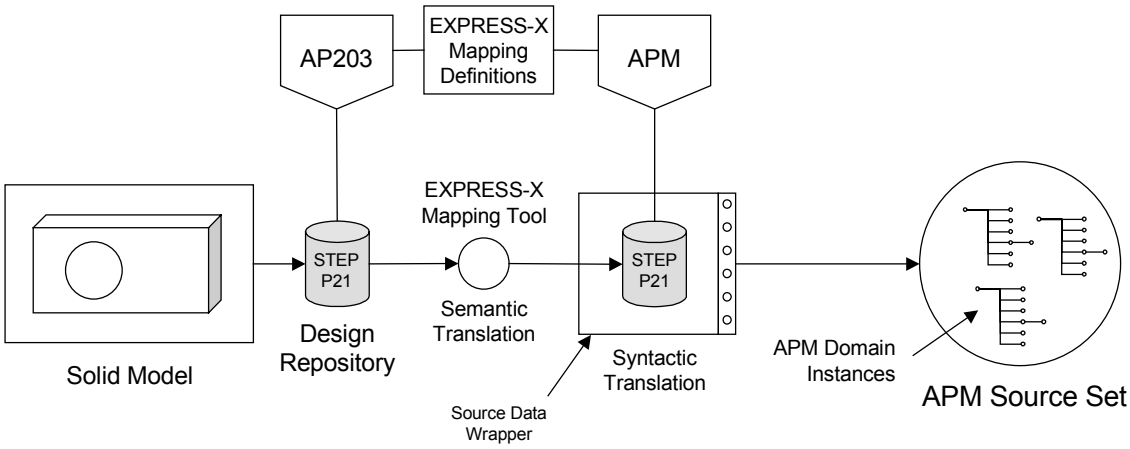


Figure 38-58: Resolving the Semantic Mismatch using EXPRESS-X

Alternatively, as shown in Figure 38-59, the solid modeler could store its data in its own native format. In this case, the solid modeler's API could be used to write a customized translator to perform the semantic translation.

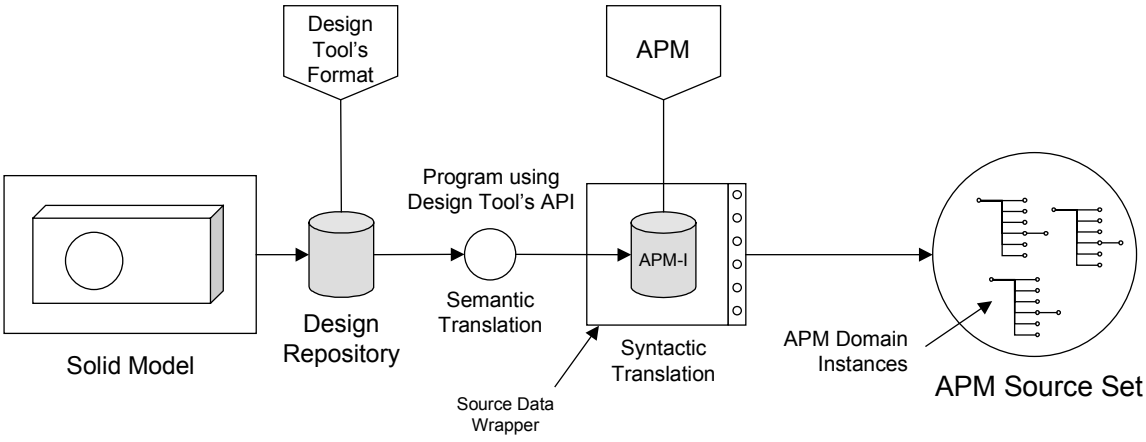


Figure 38-59: Resolving the Semantic Mismatch using the Design Tool's API

Regardless how this semantic translation is actually implemented, there must be some mechanism through which the translator can unambiguously identify the objects in the solid model that need to be translated. For example, as illustrated in Figure 38-60, there must be some way for the semantic translator to identify that attribute **`r`** of **`solid3`** corresponds to the **`radius`** of **`hole`** in the APM, or that attribute **`a`** of **`solid2`** corresponds to the **`length`** of **`plate`** in the APM.

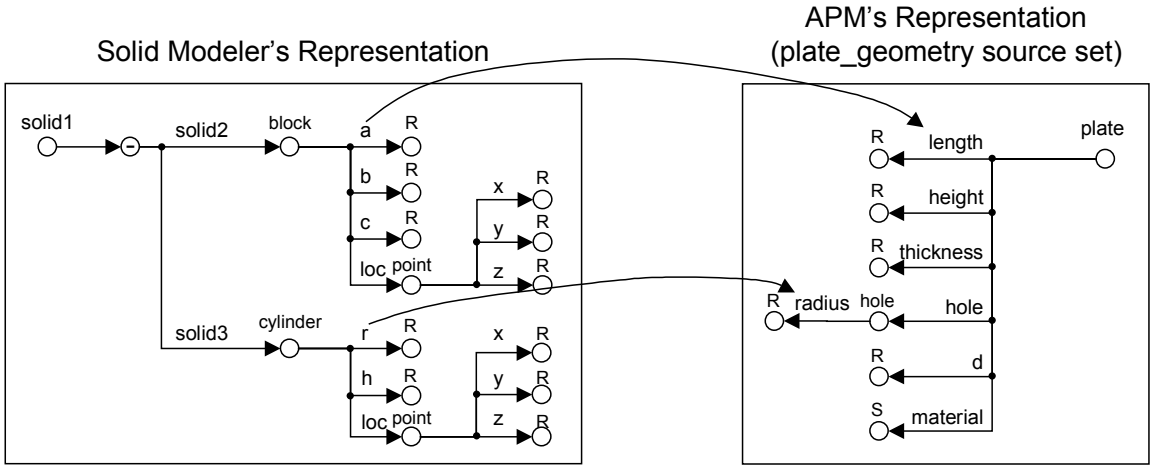


Figure 38-60: Identifying Objects to be Mapped in the Solid Model

One approach to enable this recognition (without resorting to full-fledged feature-recognition algorithms) is to manually "tag" the objects in the geometric model that are going to be translated with the names of their corresponding objects in the APM, as illustrated in Figure 38-61 (Chandrasekhar 1999). In this figure, for example, the object referenced by attribute **`solid3`** (a **`cylinder`**) in the solid modeler's representation could be tagged as "**`hole`**", and the object referenced by attribute **`solid2`** (a **`block`**) could be tagged as "**`plate`**". Next, operations that extract specific attributes from these objects could be defined using the solid modeler's API. For example, an operation called **`length()`** could be defined to return the value of attribute **`a`** of instances of **`block`**, and operation **`radius()`** to return the value of attribute **`r`** of instances of **`cylinder`**. The translator could then read the definition of the APM and determine that an attribute called **`plate.length`** and an attribute called **`hole.radius`** are needed, and call operation **`length()`** on the object labeled "**`plate`**" and operation **`radius()`** on the object labeled "**`hole`**", respectively. The obvious downside

of this approach is that a naming agreement is required between the names of the attributes in the APM and the names of the API functions used to query the solid model. Another limitation of this approach is that it makes it difficult to tag relative dimensions, because they involve more than one geometric object. For example, if the APM requires the distance between the center of the hole and one of the edges of the plate, there may not be a way to tag that distance on the solid model because it does not really correspond to any geometric entity in particular. With this approach, this distance would have to be calculated from other attributes using APM relations.

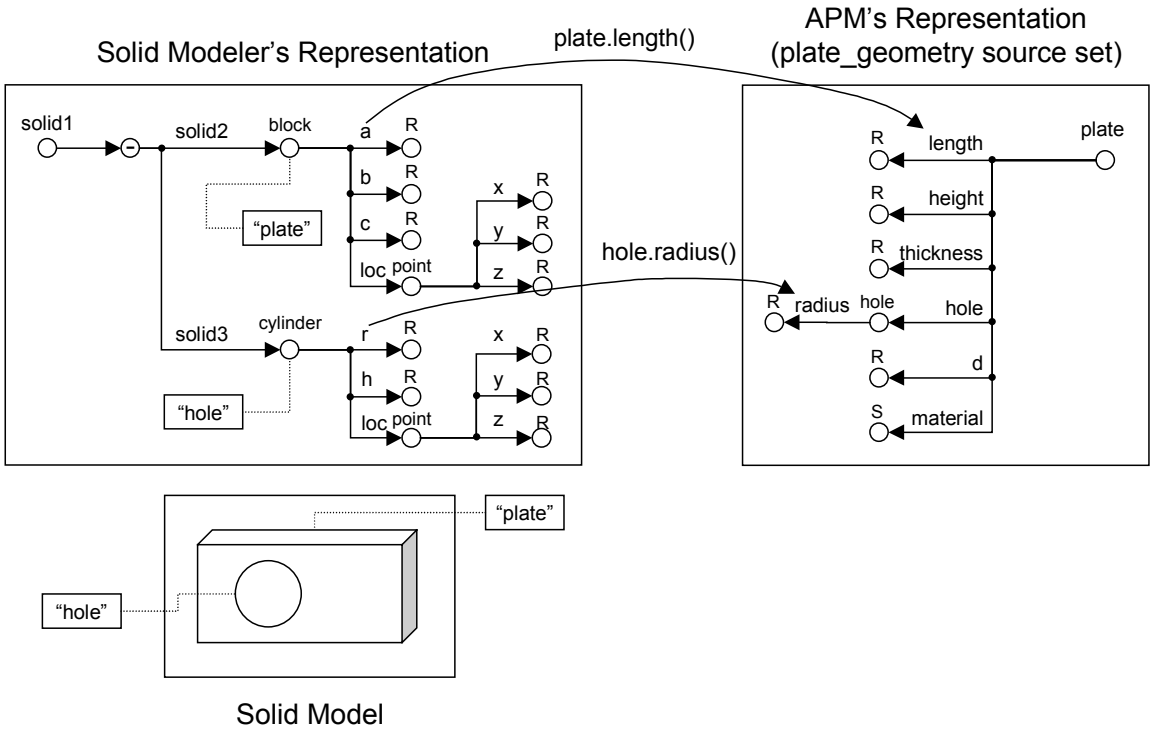


Figure 38-61: Tagging Objects in the Solid Model to Enable Semantic Translation

An alternative approach, illustrated in Figure 38-62, is to tag the *dimensions* of interest directly on the solid model. For example, the horizontal dimension of the block could be tagged as "**`plate.length`**" and the radius of the internal cylinder could be tagged as "**`hole.radius`**". The translator could then query these attributes directly by the qualified name of the dimension by using a generic operation such as

`getValueOf("plate.length")` or **`getValueOf ("hole.radius")`**. This would eliminate the need of tying the names of the API functions used to query the design model to the names of the attributes defined in the APM (as in the first approach above).

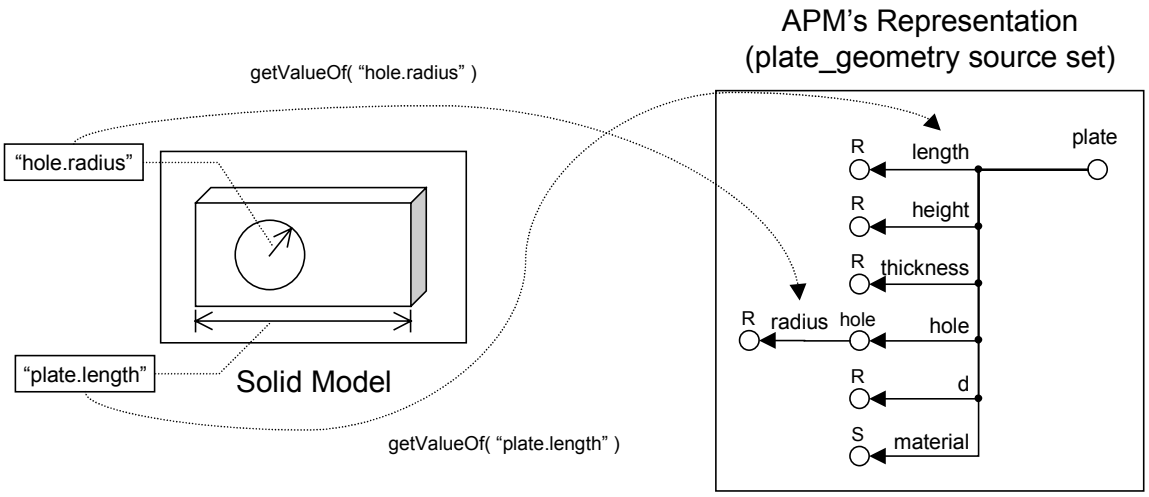


Figure 38-62: Tagging Dimensions in the Solid Model to Enable Semantic Translation

The exact mechanism through which a model is tagged will eventually depend on the capabilities of the specific design application and its API, as well as on the way in which the translator will be implemented. In any case, the common objective is to write a semantic translator that is able to find and extract information from a design model in order to satisfy the requirements of an APM.

Section 94 describes two test cases in which a model was created using a commercial solid modeling system (Dassault Systemes' CATIA) and loaded into an APM using the object tagging and the dimension tagging approaches described above.

After loading the source set data from the different design repositories, the next operation is to link this data. This operation is similar to the link APM definitions operation described in the previous subsection. The difference is that instead of linking *attributes* of APM domains, this operation links *instances* of these attributes. Figure 38-63 shows an example of the data linking operation applied on several instance of the four source sets defined above in the APM Definition File of Figure 38-48. The figure shows two instances of domain **A** (in

source set **setOne**), three instances of domain **X** (in source set **setTwo**), two instances of domain **Y** (in source set **setThree**) and four instances of domain **Z** (in source set **setFour**). The links are indicated with curved lines connecting the attributes across source sets. The result of these links is shown in Figure 38-64.

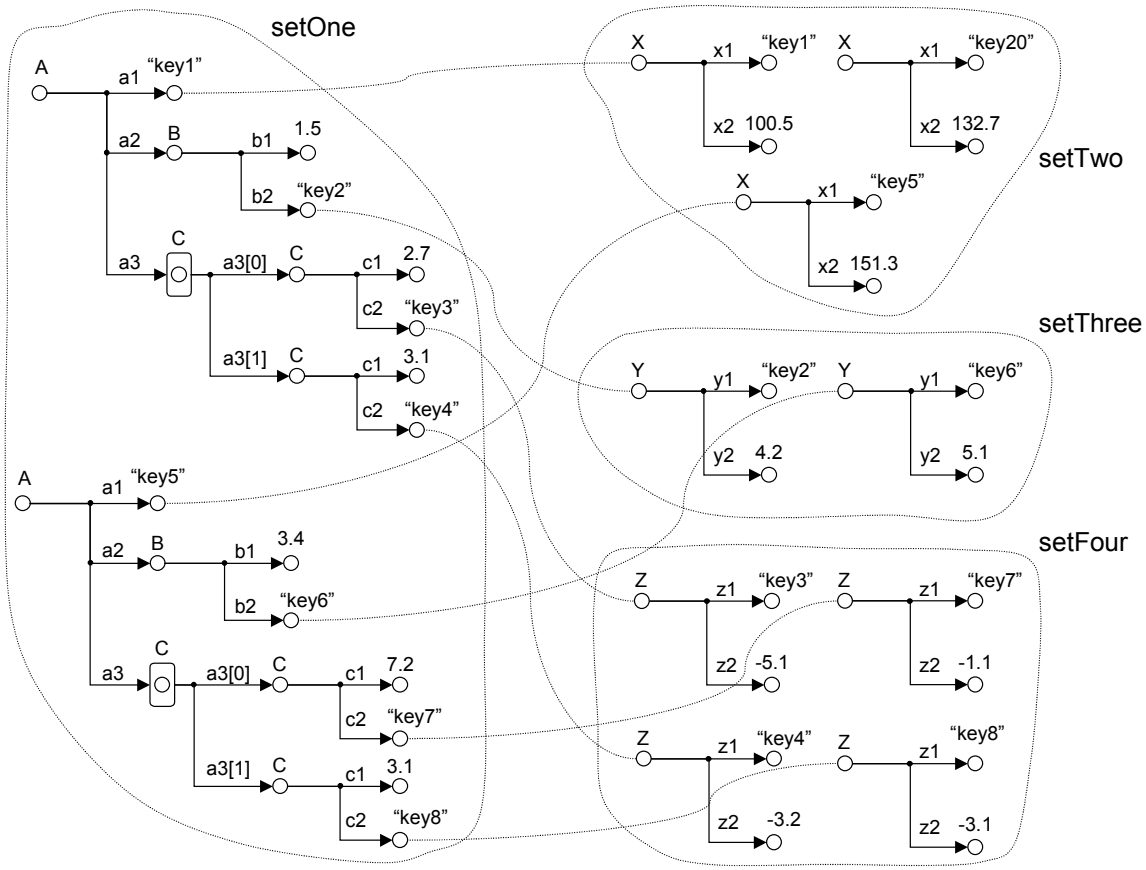


Figure 38-63: Source Set Data Linkage Example: Original Source Set Instances

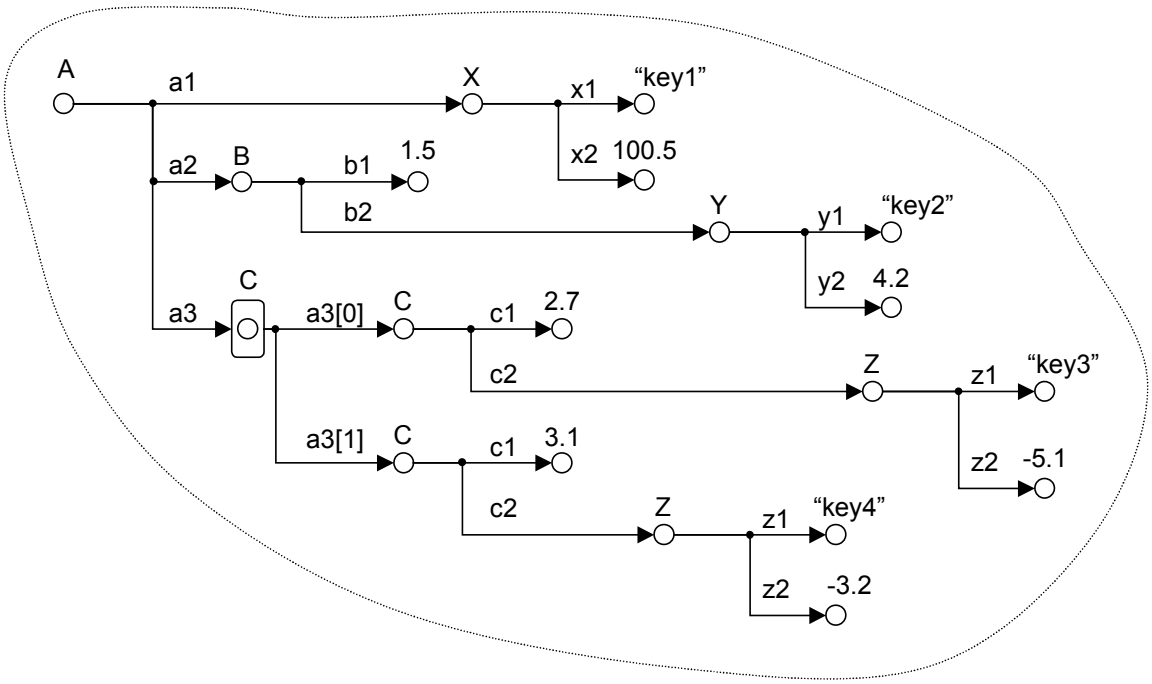


Figure 38-64: Source Set Data Linkage Example: Resulting linked Source Set Instances (only one instance of A shown)

APM Data Usage

The main purpose of loading the APM definitions and the data from different design repositories is to build an analyzable view of the product that can be *used* by APM client applications. At one level, client applications can use the APM data to query information about the *structure* of the APM itself. In this case, instances of APM Domains, APM Attributes, APM Source Set, APM Source Set Links, and APM Relations are accessed. For example, an application for browsing APM structures could request a list of the names of the attributes of domain "`flap_link`", or the relations in which attribute "`effective_length`" participates. At another level, client applications may access the *values* of design and idealized attributes defined in the APM. In this case, instances of APM Domain Instances are accessed. For example, an analysis application could request the value of the idealized attribute "`effective_length`" of a particular flap link instance in order to plug it into its analysis models. Some of these values will come directly from the design description of the product, while others will have to be derived or idealized as they are requested.

Strictly speaking, *all* operations defined in the APM Protocol can be considered to be APM usage operations (after all, they all access APM data). However, there is a group of operations that is most likely to be used by developers of APM client applications directly. Some of these operations, grouped by task, are described next.

1. Retrieving instances of a given domain: this is often the first APM usage operation performed. The result of this operation is a list of APM Domain Instances whose domain is the one requested. To illustrate this operation, consider an APM which, upon loading and linking the APM definitions and design data, contains two instances of `flap_link:` one with `part_number` equal to "FLAP-001" (Figure 38-65) and the other with `part_number` equal to "FLAP-002" (Figure 38-66).[32]

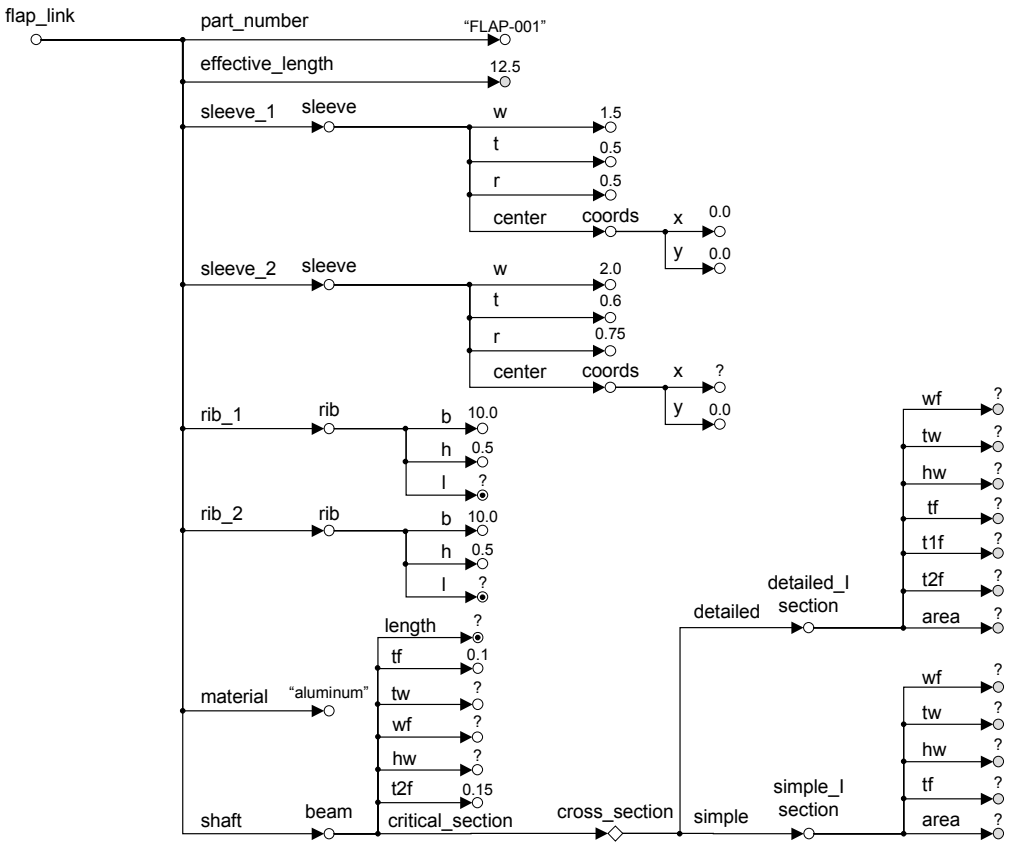


Figure 38-65: Flap Link Example: Flap Link Instance "FLAP-001"

[32]The phrase "an instance of flap_link" actually refers to an instance of APM Domain Instance (more specifically, in this case, an instance of APM Object Domain Instance) whose domain name is "flap_link".

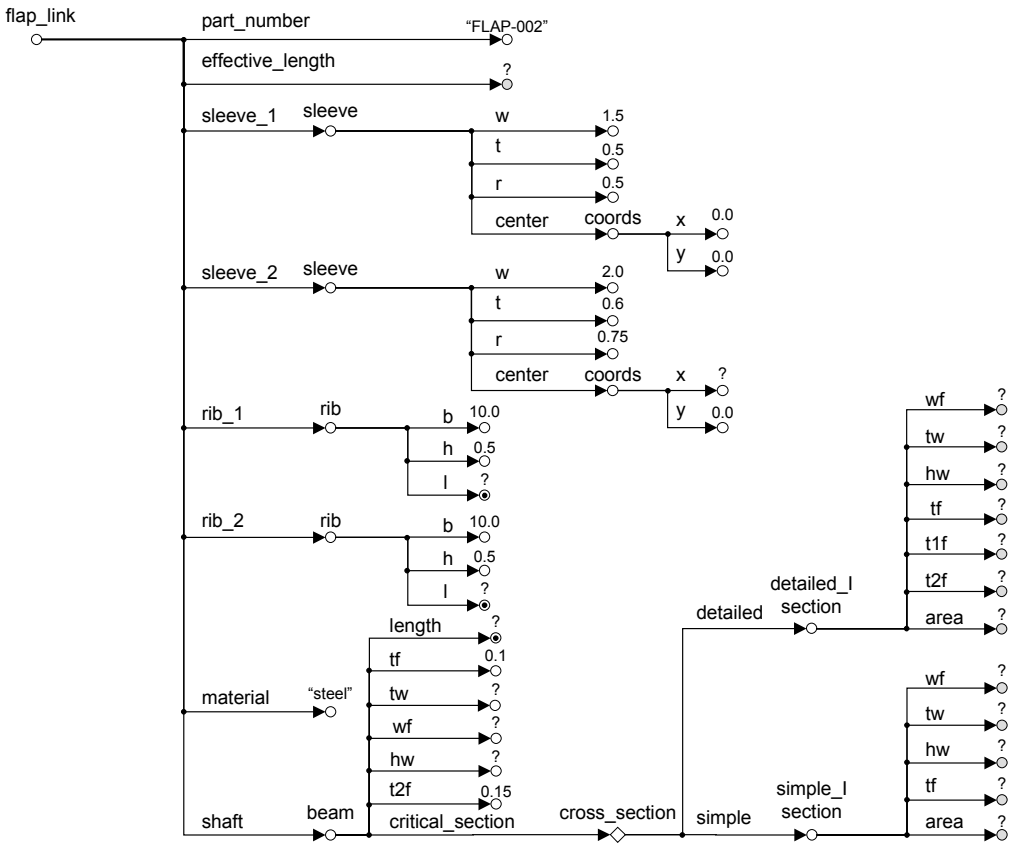


Figure 38-66: Flap Link Example: Flap Link Instance "FLAP-002"

In this example, a request for all instances of APM Complex Domain `flap_link` would return a list of two APM Complex Domain Instances ("FLAP-001" and "FLAP-002"). Individual instances of `flap_link` can then be extracted from this returned list for further manipulation.

2. Getting the value of a primitive attribute: the APM Protocol must include operations to get the values of the two types of APM Primitive Instances that may exist in an APM (namely, APM Real Instances and APM String Instances). Before retrieving their *values*, the APM Primitive *Instance* itself must be retrieved first. For this purpose, the protocol must also include operations to retrieve all types of APM Domain Instances. Combinations of these operations would be used to retrieve the primitive instance at the end of the object-attribute tree. For example, the following sequence of operations would be used to get the

value of attribute "area" in the flap link example above from a given instance of domain flap_link called **flapLinkInstance**:

1. From **flapLinkInstance**, get its "shaft" attribute (an APM Object Domain Instance of type beam).
2. From the instance obtained in 1, get its "critical_section" attribute (an APM Multi-Level Domain Instance of type cross_section).
3. From the instance obtained in 2, get its "simple" level (an APM Object Domain Instance of type simple_I_section).
4. From the instance obtained in 3, get its "area" attribute (an APM Real Instance).
5. From the instance obtained in 4, get its value (a real number).

It is important to highlight that the last operation above may have to do much more than just returning the value of the APM Real Instance. For example, if the APM Real Instance retrieved in step 4 does not have value (as it is the case in this example; notice that "area" has a question mark in Figure 38-66) then the get value operation will have to a) prepare the appropriate system of equations (using the constraint network and the values available in the APM); b) send it to an external constraint solver and c) process the results. These constraint-solving details must be handled by the get value operation and kept hidden from the programmer.

3. Setting the value of a primitive attribute: in some cases, the result of an analysis or direct input from the user will determine the value of a primitive instance. The APM Protocol must provide functions to allow setting the value of APM Primitive Instances.

4. Accessing APM structure information: some APM client applications (such as APM browsers and APM development applications) only need to access information about the *structure* of the APM. These applications only access the type of instances labeled "Model Definition Data" in Figure 38-19, and they are also known as *Generic APM Client Applications*, since they are designed to work on any valid APM. Virtually all the attributes of instances of the various constructs of the APM Representation should be accessible through operations defined in the APM Protocol. A sample of some attributes of interest from various APM constructs is:

APM Construct	Attributes of Interest
▸ APMs	Source sets, source set links, constraint network
▸ APM Source Sets	domains in set
▸ APM Domains	name
▸ ▸ APM Complex Domains	Relations, attributes
▸ ▸ ▸ APM Object Domains	local attributes, inherited attributes, supertype domain.
▸ ▸ ▸ APM Multi-Level Domains	levels
▸ ▸ APM Aggregate Domains	domain of elements, elements
▸ APM Attributes	name, domain, containing domain
▸ APM Relations	name, relation, related attributes
▸ Constraint Networks	variables, relations, nodes

5. Relaxing a relation from the constraint network: In some instances, the analyst may consider appropriate to "relax" a relation, effectively removing it from the constraint network. A relaxed relation will not be taken into account when creating the constraint system mentioned in point 3 above.

APM Data Saving

The APM Protocol must provide operations to save instances created or modified during the utilization of the APM can be saved for later use in two ways: 1) as instances conforming to the *linked* version of the APM, or 2) As instances conforming to the original *individual source sets* of the APM. Figure 38-67 shows a simple example illustrating these two options. In this example, instances of domain **`A`** from **`setOne`** are linked with instances of domain **`X`**

from **setTwo**, producing integrated instances of the augmented domain **A**. After the values of these instances are used, set or modified by the client applications they can be saved in an integrated form (in a single repository labeled "Linked APM Data" in the figure) or in separate repositories corresponding to the original source sets (labeled "setOne Data" and "setTwo Data"). Although this example shows the data being saved in STEP P21 format, in general, it can be stored in any other format.

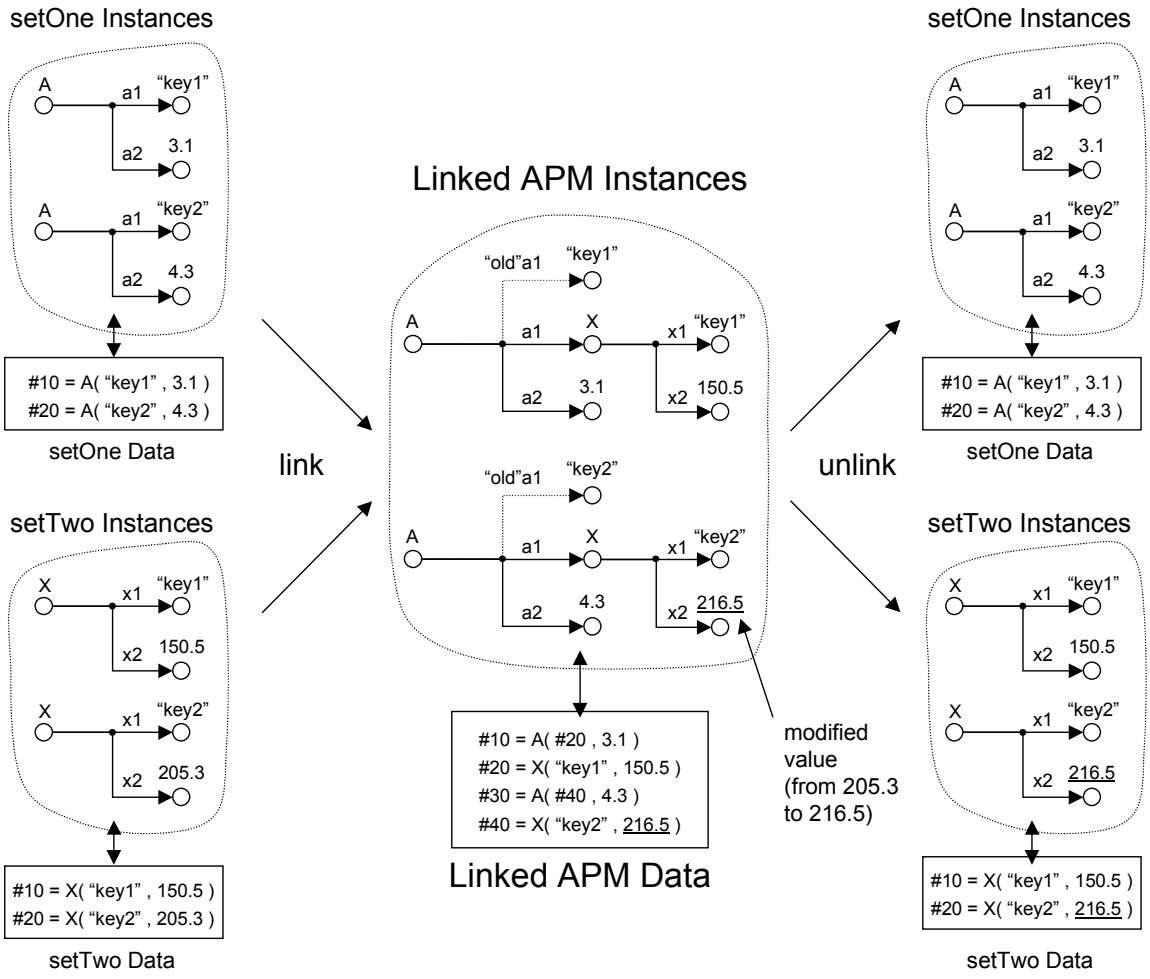


Figure 38-67: APM Data Saving Example

In order to save the APM data back to the original design repositories, an "unlink" process must take place. During this process, "foreign" instances (instances that do not belong to the

source set) are stripped from the containing instance. This is illustrated in the example of Figure 38-67 above, where the instance of domain **X** in the integrated APM belongs to **setTwo** but has been attached to an instance of domain **A** that belongs to **setOne**. Thus, when an instance of domain **A** is stored to its original design repository, the instance of set **X** attached to it is recognized as foreign and pruned. The "old" value of the attribute that was in that position before linking is restored in its place.

Potential Uses of the Mathematical APM Constructs and Operations

In Section 41, the various constructs that make up the APM Information Model were formally defined using set and graph notation and theory, and Section 58 defined conceptual operations on these constructs. These definitions provided a formal framework for the APM Representation used in the rest of the chapter to define the overall APM methodology. In the next chapters, this thesis will concentrate on evaluating the applicability of this conceptual framework and validating its usefulness with a prototype implementation and several industrial test cases.

However, an alternate path that this thesis could have taken (and probably future theses will) is to focus on further mathematical aspects of the APM Representation and define interesting properties and theorems that could be derived from it. Since the APM Representation is largely based on set and graph theory (for example, APM Domains can be represented as graphs, that is, constraint networks), there is already a solid mathematical foundation that provides a number of theorems and algorithms that such study could potentially leverage. The objective of this subsection is to provide a few examples of the concepts that could be developed by someone pursuing this alternate path[33].

Example 1: Defining operations to access construct attributes

These operations would retrieve the attributes of the constructs defined in the APM Information Model. For example, an operation called `name( )` could be defined for APM Attributes to retrieve the name of the attribute as follows:

[33] These examples are intended to provide a *flavor* of the type of concepts that could be defined with the mathematical approach, and therefore they are neither comprehensive nor fully developed.

name(a_i **)** = *domain_name*

Where:

$a_i \in \mathcal{A}$ (Definition 38-28);

domain_name is defined in Definitions 38-16, 38-18, 38-20, 38-22, and 38-24.

Another example is operation **supertype( )**:

supertype(d_i **)** = *supertype_domain*

Where:

$d_i \in \mathcal{OD}$ (Definition 38-3);

supertype_domain is defined in Definition 38-2.

Example 2: Defining complex APM operations

Formal definitions for complex APM operations (such as **linkAPMSourceSets**, **createConstraintNetwork**, or **getValue**) would effectively define the algorithms for these operations, thus facilitating subsequent implementation and optimization. For example, operation **getValue** could be expressed as:

s = **getValue(** ri_i **)**

Where:

$ri_i \in \mathcal{RI}$;

s is an integer indicating the number of solutions found.

Operation **getValue( )** assigns the first positive solution found (or the first solution, if there are no positive solutions) to attribute v of ri_i.

The algorithm for this operation could be described as:

1. constraintNetworkVariableNode = **getNode(name(** ri_i **)** , constraintNetwork **)**
2. connectedRelations, { cr_1 , cr_2 , … , cr_n } = **getConnectedRelations(** constraintNetworkVariableNode , constraintNetwork **)**
3. connectedVariables, { cv_1 , cv_2 , … , cv_m } = **getConnectedVariables(** constraintNetworkVariableNode , constraintNetwork **)**
4. connectedInstances, { ci_1 , ci_2 , … , ci_p } = **getConnectedInstances(** ri_i , connectedVariables **)**
5. connectedInstancesWithValues, { ci_i | ci_i ∈ connectedInstances **: hasValue(** ci_i **)** = true }
6. results, { r_1 , r_2 , … , r_s } = **solveFor(** ri_i , connectedRelations , connectedInstancesWithValue **)**
7. **setValue(** ri_i , r_j **)** where r_j = first positive element of { r_1 , r_2 , … , r_s } (if any) or r_j = r_1 (if there are no positive elements)
8. return s

Where:

constraintNetworkVariableNode ∈ $\mathcal{CNN}$;

constraintNetwork ∈ $\mathcal{CN}$;

cr_i ∈ $\mathcal{CNR}$;

cv_i ∈ $\mathcal{CNV}$;

ci_i ∈ $\mathcal{PI}$;

r_i ∈ $\mathbb{R}$

Operations **getNode**, **getConnectedRelations**, **getConnectedVariables**, **getConnectedInstances** above (not defined) could be based on search algorithms widely

available for graphs (Thomas 1992). Operation **`solveFor`** could use the services of an external constraint solver to find a solution for the constraint system.

Example 3: Derive properties and theorems

Given the mathematical definition of a construct, several properties and theorems of interest could be derived from it that could help predict the characteristics of a given APM.

An example of a theorem that could be stated from the definition of Constraint Networks and operation **`getValue`** could be of the form:

> Example theorem 1: "Given a constraint network **N**, the minimum number of calls to operation **`solveFor`** that will have to be performed in order to solve for *all* its variables is equal to the number of connected components of **N**."[34]

An example of a useful property that could be defined is the computational "cost" of an idealization. Such property would serve as a metrics to help an analyst decide among several levels of idealization fidelity. For example, Figure 38-68 illustrates two idealized attributes in a constraint network, one more expensive than the other based on the number of relations and attributes connected to it.

[34] A connected component of a graph is the largest set of vertices in which each vertex in the set is reachable from any other vertex in the same set. If the graph is disconnected, it will have more than one connected component.

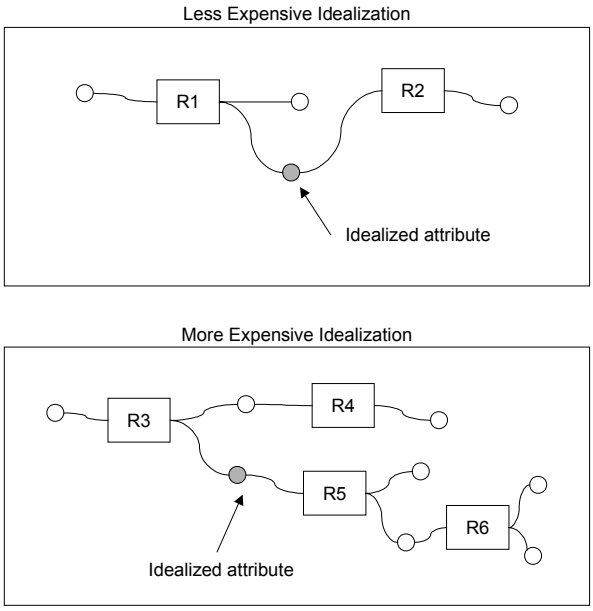


Figure 38-68: Idealization Cost Property

Another property that could be defined is the "sensitivity" of an APM Primitive Attribute. Such property would provide a measure of the impact that changing the value of the attribute has on the rest of the constraint network. Such metrics would help an analyst decide which attributes he or she can change without affecting the others too much. For example, in Figure 38-69 the analyst has the choice of adjusting the value of attributes **a1** and **a2** in order to reach a target value of **a3=5.5**. In this example, changing adjusting the value of **a2** has less impact over adjusting the value of **a1** (in other words, **a1** is "more sensible" than **a2**), since there are less attributes connected to **a2** than to **a1**.

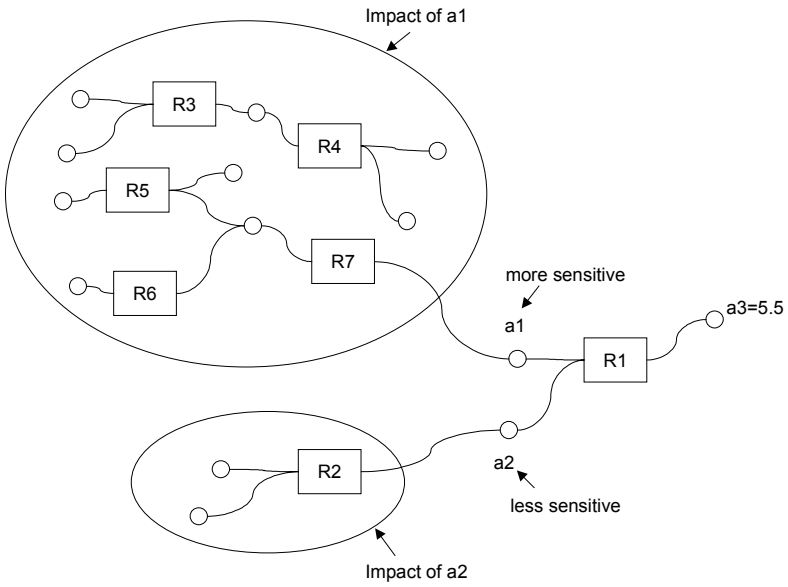


Figure 38-69: Sensitivity Property

CHAPTER 5

PROTOTYPE APM REPRESENTATION IMPLEMENTATION

Chapter 38 laid the theoretical foundation for the APM Representation by formally defining the fundamental APM concepts and specifying the structure and operations of the APM. As mentioned in that chapter, the APM Representation can then be implemented in specific information modeling and programming languages to be used in the development of design-analysis integration applications. As shown in Figure 64-1, this chapter presents one such implementation - prototyped by the author – whose application will be demonstrated in Chapter 83 with four test APM definitions and four test APM client applications.

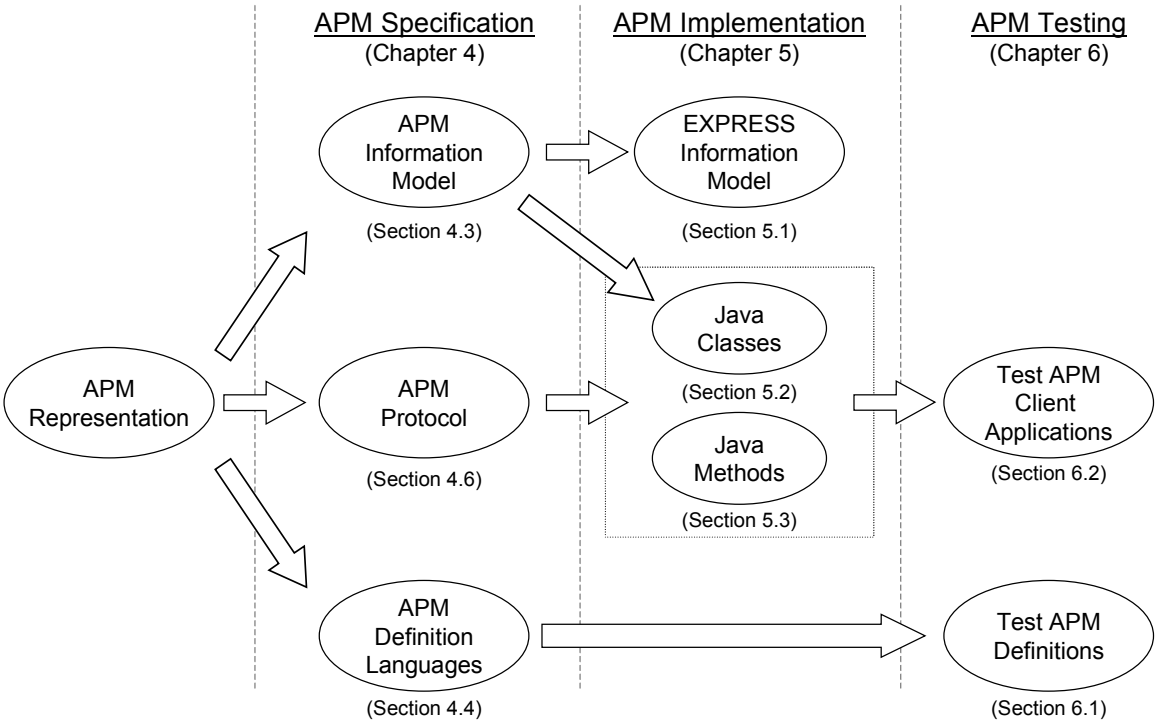


Figure 64-1: APM Representation Implementation

The main objective of developing a prototype implementation of the APM was to be able to test the functionality of the APM concepts and evaluate, with several test cases, how well the constructs defined in the APM Information Model and the operations defined in the APM Protocol satisfy the requirements of design-analysis integration. In addition, the implementation process itself and the practical issues introduced by the test cases helped refine the APM Representation, by revealing the need for additional operations and constructs that had not been considered during the conceptualization stage.

This chapter is organized as follows: Section 65 presents an implementation of the APM Information Model using the EXPRESS information modeling language. Next, Section 76 presents an implementation of the APM Information Model using the Java programming language. Finally, Section 77 presents a Java implementation of the operations of the APM Protocol.

APM Information Model Implementation in EXPRESS

Chapter 38 provided the theoretical definitions of the various constructs that make up the APM Information Model (Section 41). As illustrated in Figure 64-2, this section will describe how these theoretical definitions were translated into a specific information modeling language (EXPRESS).

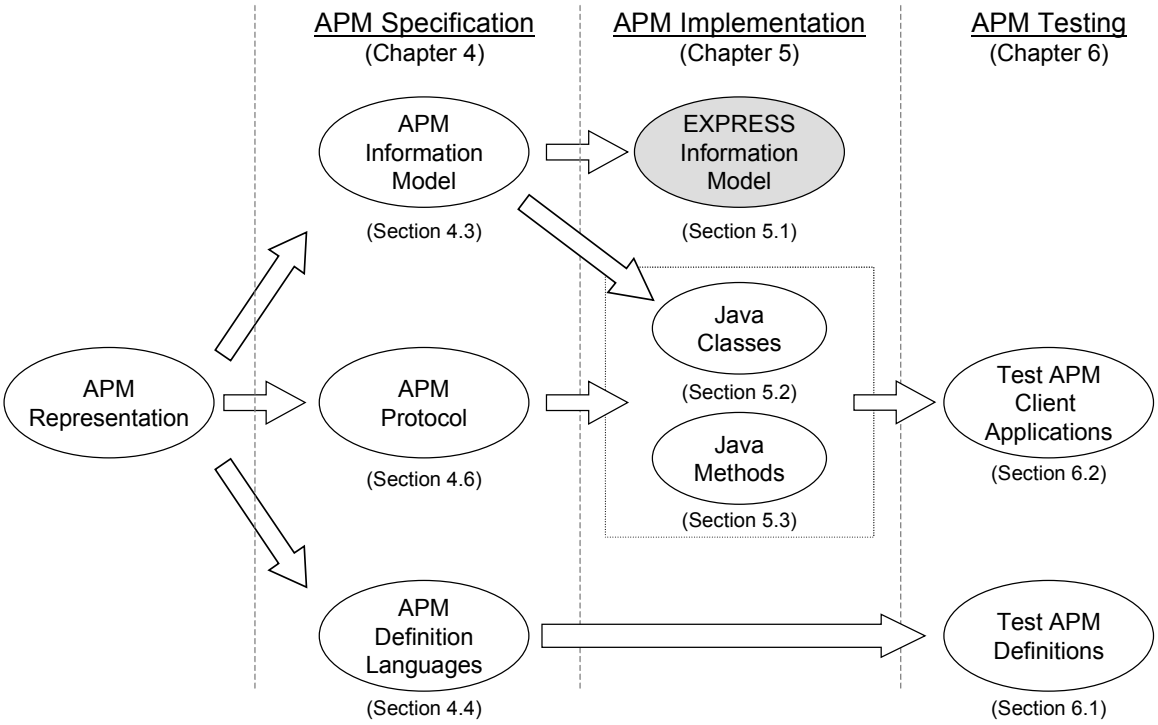


Figure 64-2: APM Information Model Implementation in EXPRESS

EXPRESS (ISO 10303-11 1994; Schenck and Wilson 1994; Wilson 1991) was selected as the information modeling language for this implementation for the following reasons:

1. It provides the neutral mechanisms for describing and exchanging product information;
2. There is a large number of EXPRESS-based development tools to aid in the development of STEP applications available both commercially and publicly (Denno 1997; National Institute of Standards and Technology (NIST) 1999; Spooner 1993; STEP Tools Inc 1997a; STEP Tools Inc 1997b; STEP Tools Inc 1997c; Wilson 1998);
3. It has an object-oriented flavor;
4. It is one of the few information modeling languages that has both a formal lexical form (EXPRESS) and a graphical form (EXPRESS-G). Being a textual language means that computer-based parsers, compilers and report writers are available for the

language. In other words, it is computer-interpretable (Schenck and Wilson 1994; Wilson 1991); and

5. It enjoys a growing international acceptance by industry, government and academia.

Since there is a closer correspondence between information modeling language and programming language constructs than there is between mathematical and programming language constructs, having the APM Information Model described in terms of an information modeling language facilitates its implementation in a programming language. For example, EXPRESS data models can be compiled to automatically generate C++ or Java classes for each entity in the model, as well as basic "get" and "put" functions for all attributes. Though possible, it would be more difficult to implement the APM Information Model in a programming language directly from their mathematical definitions. In addition, most information modeling languages come in graphical form or have a companion graphical form, aiding communication and documentation of the APM Information Model. Of course, the choice of information modeling language and the choice of programming language are not totally independent from each other; a bad combination of choices could make implementation more difficult than it should be. For example, it is easier to implement a model described with an object-oriented information modeling language (such as EXPRESS, UML - (Booch, Jacobson et al. 1998; Fowler 1998; Si Alhir 1998), or OMT - (Rumbaugh, Blaha et al. 1991)) with an object-oriented language (such as C++, Smalltalk or Java).

In order to understand how the mathematical definitions presented in the previous chapter were mapped into EXPRESS, it is useful to understand how EXPRESS and set theory are related. In essence, EXPRESS can be viewed as a set declaration language (Schenck and Wilson 1994; Wilson 1996); an EXPRESS entity declaration effectively defines a set, and the attributes of this entity the n-tuples that define each member of the set. To illustrate this, consider the following two set definitions:

$A = \{ (a_1 , a_2) \mid a_1 \in \mathbb{R} , a_2 \in B \}$, and

$B = \{ (b_1 , b_2) \mid b_1 \in \mathbb{S} , b_2 \in \mathbb{Z} \}$

Which state that elements of set **A** are 2-tuples of the form ($\mathbf{a_1}$, $\mathbf{a_2}$), where $\mathbf{a_1}$ is a real and $\mathbf{a_2}$ is an element of set **B**, and elements of set **B** are 2-tuples of the form ($\mathbf{b_1}$, $\mathbf{b_2}$), where $\mathbf{b_1}$ is a string and $\mathbf{b_2}$ is an integer.

A *specific element* of **A**, called $\mathbf{A_i}$, may also be defined as:

$A_i = (a_1 , a_2)$

Where:

$a_1 \in \mathbb{R}$;

$a_2 \in B$.

And a specific element of **B**, called $\mathbf{B_i}$, may be defined as:

$B_i = (b_1 , b_2)$

Where:

$b_1 \in \mathbb{S}$;

$b_2 \in \mathbb{Z}$.

The following two EXPRESS definitions are equivalent to the two set declarations above:

```
ENTITY A;
   a1 : REAL;
   a2 : B;
END_ENTITY;

ENTITY B;
   b1 : STRING;
   b2 : INTEGER;
END_ENTITY;
```

Another point that requires clarification - before getting into the descriptions of the particular groups of entities - is the use of *inheritance* in EXPRESS models. Since EXPRESS supports inheritance (one of the most significant features of an object-oriented model) common attributes of some entities in the APM Information Model are grouped into a

common supertype entity. For example, in Figure 64-3, attribute "`domain_name`" is shared among all APM Domains (this is consistent with Definitions 38-2, 38-4, 38-7, 38-10, and 38-12). Hence, "`domain_name`" is defined as an attribute of entity `apm_domain`, which is defined as the supertype of all APM Domains. Entities such as `apm_domain` - created solely for gathering attributes that are shared among its subtypes - are normally declared *abstract*[35] (meaning that there will be no instances of these entities).

The subsections that follow describe the entities of the APM Information Model that result from mapping the APM definitions of the previous chapter into EXPRESS. Although the APM Information Model is actually a single EXPRESS schema, its entities will be presented in the same groups in which they were introduced in the previous chapter, namely:

1. APM Domain Entities;
2. APM Attribute Entities;
3. APM Domain Instance Entities;
4. APM Source Set Entities;
5. APM Source Set Link Entities;
6. APM Relation Entities; and
7. Constraint Network Entities.

Three additional groups - not defined in the previous chapter – have been added to the APM Information Model for implementation reasons:

1. APM Interface Entities;
2. APM Source Set Data Wrapper Entities; and
3. APM Solver Wrapper Entities.

In general, there is a one-to-one correspondence (in naming and meaning) between the APM constructs defined mathematically in the previous chapter and the entities of the EXPRESS APM Information Model introduced in this section. Therefore, it is quite easy to understand

[35] In EXPRESS-G diagrams, abstract entities are indicated by the symbol (ABS) next to the entity name. In EXPRESS, they are indicated by the keyword `ABSTRACT`.

the connections between the two. To highlight these connections, each APM entity is labeled in the EXPRESS-G diagrams of Figures 64-3 through 64-12 with the numbers of the definitions to which it corresponds[36]. Since the basic APM constructs have been already formally defined and explained in the previous chapter, the following subsections will only provide a brief overview of each group of entities, only explaining in more detail any new entities, new attributes, name changes or deviations from these definitions.

The complete EXPRESS APM Information Model is included in Appendix J, and the corresponding EXPRESS-G diagrams are included in Appendix K. Subsection 55 introduces most of the symbols used in these EXPRESS-G diagrams. The lexical EXPRESS definitions (when read along the EXPRESS-G diagrams) should be simple enough for the reader who is not an expert in EXPRESS to understand. However, additional clarification will be provided for those portions of the EXPRESS syntax that are less obvious. For a complete description of both the lexical EXPRESS language and the EXPRESS-G nomenclature the reader may refer to (ISO 10303-11 1994), (Schenck and Wilson 1994) and Appendix A.

APM Domain Entities

Figure 64-3 is an EXPRESS-G diagram showing the entities that form the APM Domain Entities group.

As the diagram shows, all APM Domains are subtyped from the abstract entity `apm_domain`. Two new attributes[37] were defined in this entity: the optional attribute `domain_description`[38] and the required attribute `source_set`[39]. As the name suggests, `domain_description` is a string that holds a description of the domain. Attribute `source_set`

---

[36] A given EXPRESS entity can be viewed as the declaration of two things: 1) a set *as a whole*, and 2) the *individual elements* of the set. In the example given a few paragraphs above, EXPRESS entity A defines 1) the set A, and 2) the 2-tuple (a_1,a_2) that defines an element A_i of set A. For this reason some entities in the APM Information Model refer to two or more definitions from the previous section. For example, in Figure 64-3, entity `apm_object_domain` refers to both the definition of a individual APM Object Domain (od_i – Definition 38-2) *and* the entire Set of APM Object Domains (OD – Definition 38-3)

[37] By "new attribute" it is meant an attribute that is not defined in the previous chapter.

[38] Optional attributes are represented with a dashed line in EXPRESS-G diagrams and with the keyword `OPTIONAL` in lexical EXPRESS.

[39] Required attributes are represented with a solid line in EXPRESS-G diagrams. No special keywords are used in lexical EXPRESS to indicate a required attribute.

points to the source set to which the domain belongs. This attribute is actually redundant; the list of source sets of a given APM (see Definition 38-69) could be transversed in order to find the source set to which a given domain belongs. However, a direct reference *from* the domain *to* the source set would prevent a search algorithm from having to transverse a potentially long list of source sets and domains (efficiency improvements such as this are the purpose of most of the new attributes added to the APM Information Model during implementation).

Entities `apm_object_domain` and `apm_multi_level_domain` are further subtyped from abstract entity `apm_complex_domain`. Both entities share the attribute `local_relations`, which corresponds to the list of APM Relations in Definitions 38-2 and 38-4. Attribute `local_attributes`[40] in entity `apm_object_domain` corresponds to the list of APM Attributes in Definition 38-2. Attribute `levels` in entity `apm_multi_level_domain` corresponds to the list of levels in Definition 38-4.

Entities `apm_complex_aggregate_domain` and `apm_primitive_aggregate_domain` are subtyped from abstract entity `apm_aggregate_domain`.

Attribute `supertype_domain` of entity `apm_object_domain` provides the means for defining inheritance hierarchies between APM Object Domains. Following the object-oriented paradigm, a given APM Object Domain inherits the attributes and the relations of its parent. Hence the reason for the prefix "local" in attributes `local_relations` and `local_attributes` – to differentiate attributes that are local to the domain from those that are inherited from a supertype.

[40] Do not confuse the attributes of an APM Object Domain with the attributes of an EXPRESS entity. For example, "`length`" is an attribute of an APM Object Domain called "`plate`", whereas "`domain_name`" is an attribute of entity `apm_domain`.

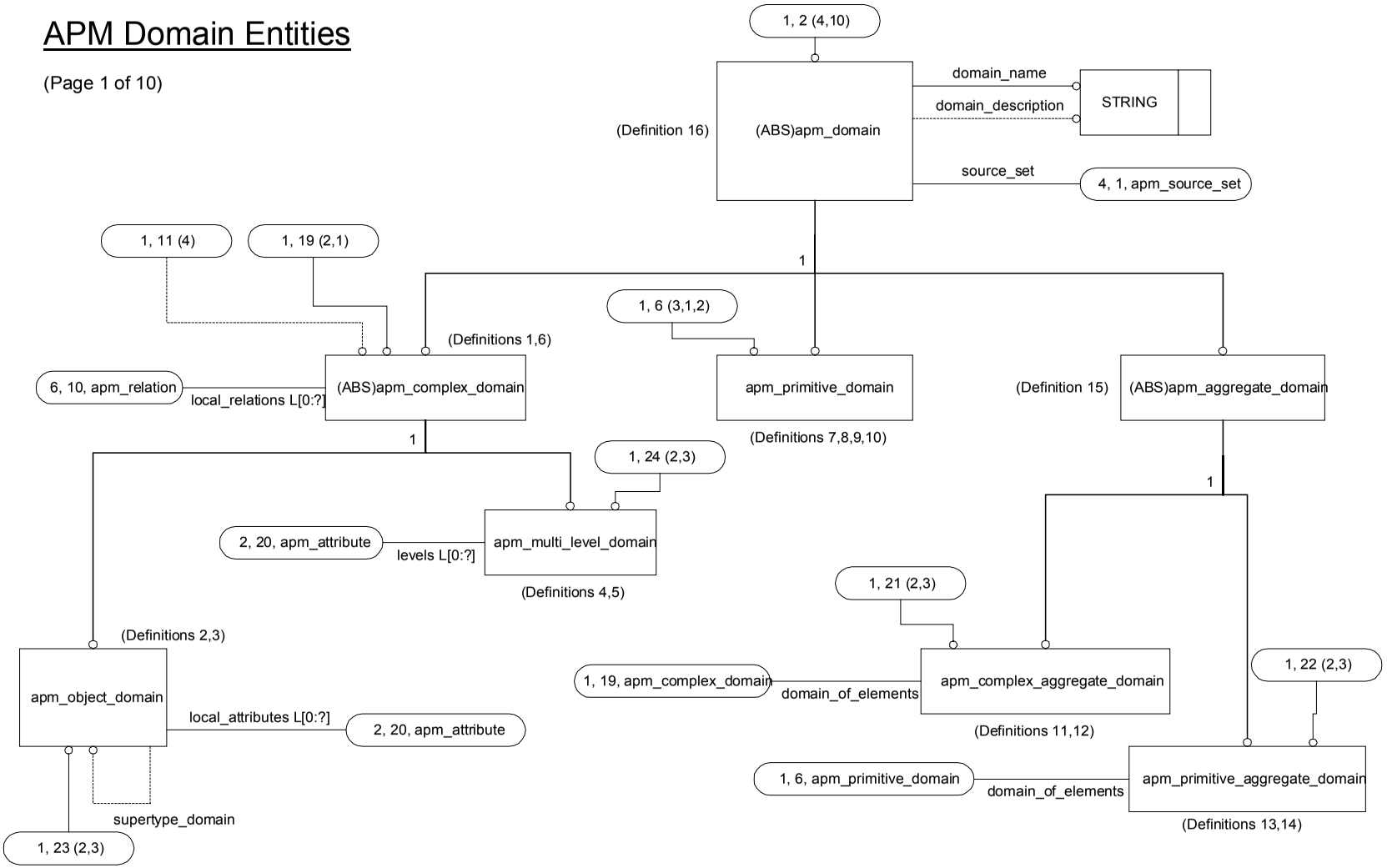


Figure 64-3: APM Domain Entities

As it can be inferred from the fact that attribute `supertype_domain` only points to *one* APM Domain, the APM model only allows *single inheritance* (that is, a class can only have one `supertype_domain`)[41]. This is mainly to avoid the problems associated with multiple inheritance, such as loss of conceptual and implementation simplicity, and conflicts among inherited definitions that must be resolved in implementations (such as delegation, inheritance and delegation, and nested generalization - see Rumbaugh, Blaha et al. 1991). It also facilitates the APM implementations in programming languages that do not support multiple inheritance (such as Java (Deitel and Deitel 1998; Deitz 1996; Dragan 1997; Flanagan 1997; Sun Microsystems 1998)).

APM Attribute Entities

Figure 64-4 is an EXPRESS-G diagram showing the entities that form the APM Attribute Entities group. Notice that the structure of this diagram resembles the structure of the previous diagram (Figure 64-3) corresponding to the APM Domain entities. This is because there is a type of APM Attribute for each type of APM Domain (for example, the domain of an APM Object Attribute is an APM Object Domain, and so on).

All APM Attributes are subtyped from the abstract entity `apm_attribute`. Two new attributes were added in this group. The first attribute is the optional attribute `attribute_description`, added to entity `apm_attribute`, and the second is the required attribute `category`, added to `apm_primitive_attribute`. Attribute `category` is an integer that can be used to categorize primitive attributes according to any arbitrary criterion (by assigning the same integer to attributes of the same category). In this work, attribute `category` is used to group primitive attributes into product, essential and idealized attributes (see Definitions 38-61, 38-62, and 38-64).

[41] EXPRESS, on the other hand, does allow multiple inheritance

APM Attribute Entities

(Page 2 of 10)

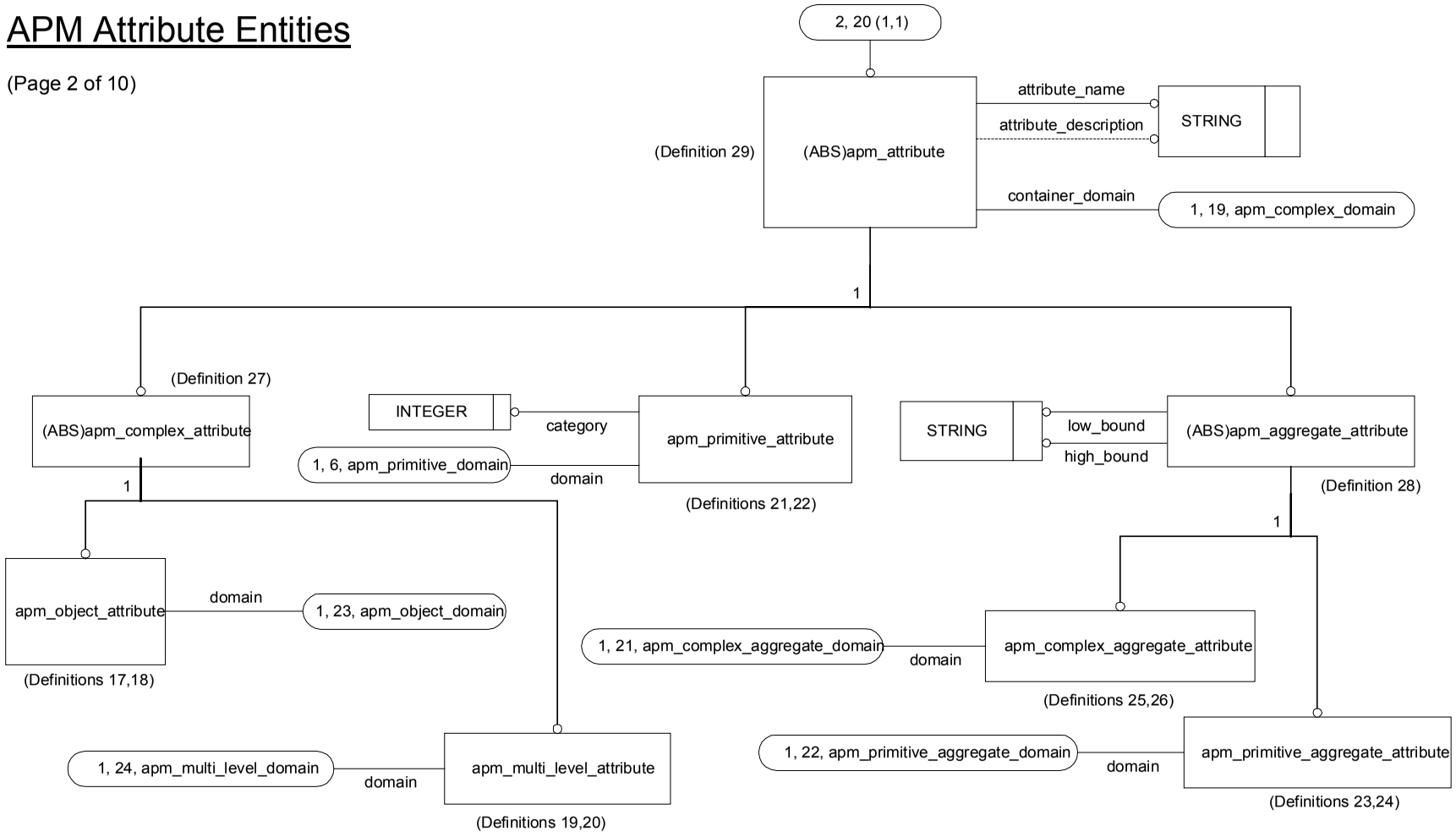


Figure 64-4: APM Attribute Entities

Entities apm_object_attribute and apm_multi_level_attribute are subtyped from abstract entity apm_complex_attribute. Entities apm_complex_aggregate_attribute and apm_primitive_aggregate_attribute were subtyped from abstract entity apm_aggregate_attribute, which contains attributes low_bound and high_bound, common to both (see Definitions 38-22 and 38-24).

APM Domain Instance Entities

Figure 64-5 is an EXPRESS-G diagram showing the entities that form the APM Domain Instance Entities group. The structure of this diagram also resembles the structure of the diagram for APM Domains (Figure 64-3), since there is also a type of APM Domain Instance for each type of APM Domain, depending on the type of APM Domain the APM Domain Instance is instantiating (for example, an APM Object Domain Instance, as the name suggests, is an instance of an APM Object Domain).

All APM Domain Instances are subtyped from the abstract entity apm_domain_instance. Entities apm_object_domain_instance and apm_multi_level_domain_instance are further subtyped from the abstract entity apm_complex_domain_instance. Entities apm_real_instance and apm_string_instance are subtyped from the abstract entity apm_primitive_domain_instance. And entities apm_complex_aggregate_domain_instance and apm_primitive_aggregate_domain_instance are subtyped from the abstract entity apm_aggregate_domain_instance.

Three new attributes were added to all APM Domain Instances (in their common supertype entity apm_domain_instance): attribute_name (required), contained_in (optional) and element_of (optional). If an APM Domain Instance is an instance of one of the attributes (or levels) of an APM Complex Domain Instance (see Definitions 38-34 and 38-37), then attribute contained_in points to this containing APM Complex Domain Instance. In this case, attribute_name contains the name of the attribute of which this instance is an instance. This name must correspond to a name already defined in the domain referred to by contained_in (that is, must be a *valid* attribute name).

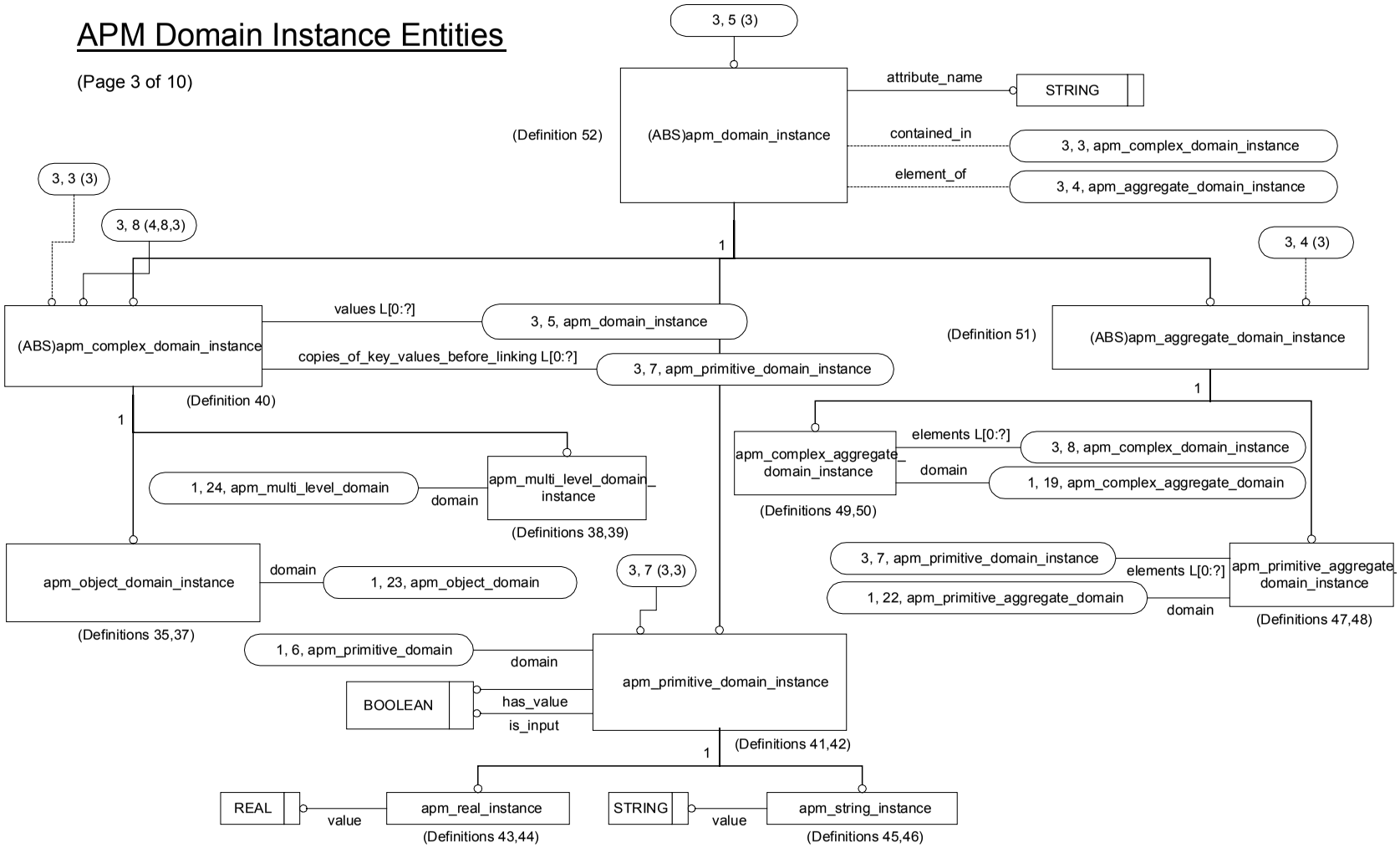


Figure 64-5: APM Domain Instance Entities

On the other hand, if an APM Domain Instance is one of the element instances of an APM Aggregate Domain Instance (see Definitions 38-46 and 38-48), then attribute `element_of` points to this containing APM Aggregate Domain Instance. When this is the case, `attribute_name` contains the same attribute name of the `apm_aggregate_instance` with the element number appended to it (for example, if "layup" is the attribute name of the containing aggregate, "layup[2]" is the attribute name of the third element of this aggregate instance).

Finally, an APM Complex Domain Instance may also exist as an independent instance (that is, not contained by any other instance). In this case, both `contained_in` and `element_of` are empty. Attribute `attribute_name` does not really have meaning in this case and, by convention, a name such as "root" may be used to indicate that this instance is not contained by any other.

Entity `apm_complex_domain_instance` has a new required attribute called `copies_of_-key_values_before_linking`. This attribute plays an important role in the "unlink" operation (when the APM is split back into its original source sets) described in Subsection 0. Attribute `values` of abstract entity `apm_complex_domain_instance` corresponds to the list of attribute instances (in the case of APM Object Domain Instances – see Definition 38-34) or to the list of level instances (in the case of APM Multi-Level Domain Instances – see Definition 38-37).

Entity `apm_primitive_domain_instance` has two new required boolean attributes: `has_value` and `is_input`. Attribute `has_value` is set to true when the instance holds a value and `is_input` is set to true when the value is considered an input in a given relation. These attributes are used by the constraint-solving algorithm to determine which values need to be solved by the constraint solver and which do not, and their roles will be explained in more detail in Subsection 81.

Attribute `elements` in entities `apm_primitive_aggregate_domain_instance` and `apm_complex_aggregate_domain_instance` corresponds to the list of element instances in Definitions 38-46 and 38-48.

APM Source Set Entities

Figure 64-6 is an EXPRESS-G diagram showing the only entity of the APM Source Set Entities group: `apm_source_set`. Attribute `domains_in_set` contains the list of APM Domains that are members of the set (see Definition 38-52). Attribute `set_instances` contains a list of APM Complex Domain Instances that belong to the set. Attribute `data_repository_name` is a new string attribute that contains the name of the file or data repository from which the instances are read. Attribute `source_set_data_wrapper`, also a new attribute, points to the APM Source Data Wrapper Object assigned to the source set. As introduced in Subsection 60, and explained in more detail in Subsection 79, source set data wrapper objects read design data in a particular format and put it into a neutral form that the APM understands. Attribute `root_domain` is a new attribute that points to a domain belonging to the list `domains_in_set` that is considered the root domain of the source set. Only instances of the root domain (or any of its subtypes) are read and added to the list of complex instances in `set_instances` during the data loading operation (see Subsection 79).

APM Source Set Entities

(Page 4 of 10)

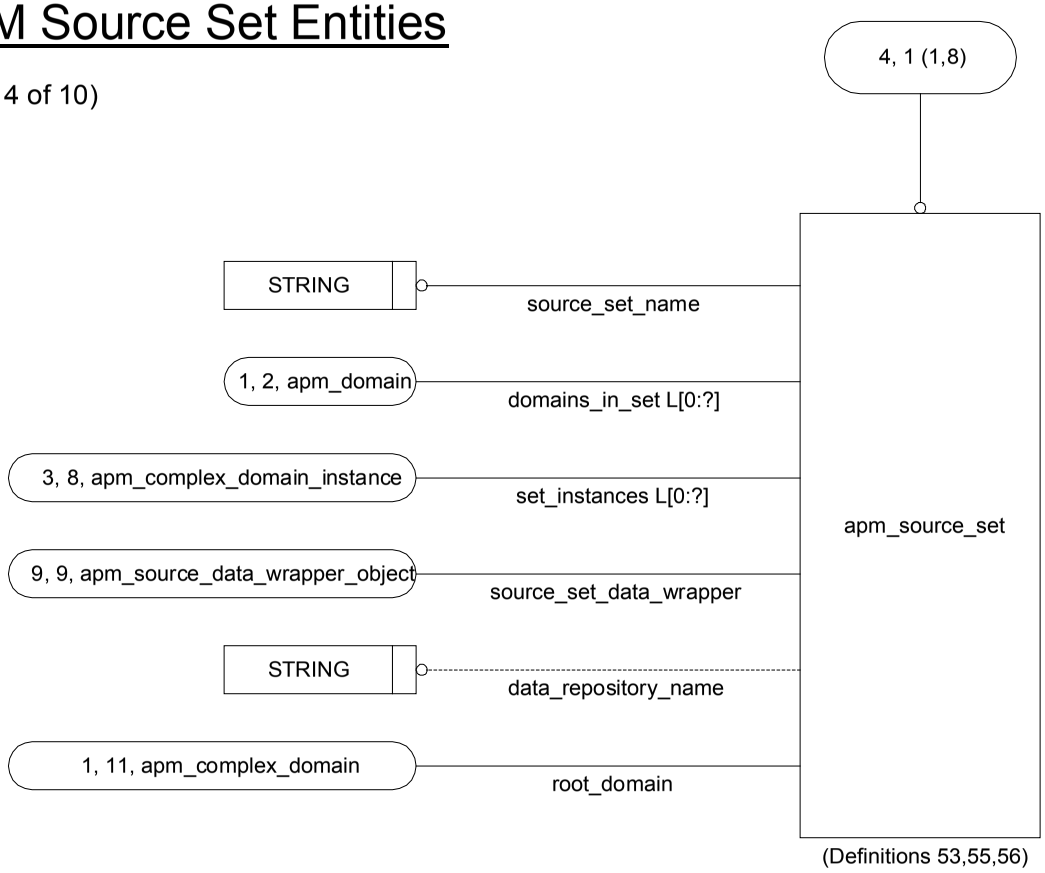


Figure 64-6: APM Source Set Entities

APM Source Set Link Entities

Figure 64-7 is an EXPRESS-G diagram showing the only two entities that form the APM Source Set Link Entities group: `apm_source_set_link` and `apm_source_set_link_attribute`. Entity `apm_source_set_link` corresponds to the simplest of the three versions of the definition of APM Source Set Link (Definition 38-59), which only requires the two key attributes (`key_attribute_1` and `key_attribute_2`) and a logical operator (`logical_operator`) to compare them. Although Definition 38-59 specifies that the key attributes be APM Primitive Attributes, this was implemented in the APM Information Model in a slightly different manner. Instead of pointing to an APM Primitive Attribute, attributes `key_attribute_1` and `key_attribute_2` of `apm_source_set_link` point to an `apm_source_set_link_attribute`, which in turn has an attribute called `full_attribute_name`, which is a list of strings. This list of strings represents the full name of the attribute. The full name of the attribute is constructed with the names of all the attributes that contain the attribute in question – all the way up to the top or "root" attribute – separated by dots. Instead of "root" as the first attribute name in the list, the domain name of the top domain is used. For example, if a domain **A** has an attribute called **a1** of type **B**, which in turn has an attribute called **b1** of type **C**, which in turn has a string attribute called **c1**, the full attribute name of an instance of **c1** would be **A.a1.b1.c1**. Thus, in this example, `full_attribute_name` would be the list of strings { "A" , "a1" , "b1" , "c1" }.

APM Source Set Link Entities

(Page 5 of 10)

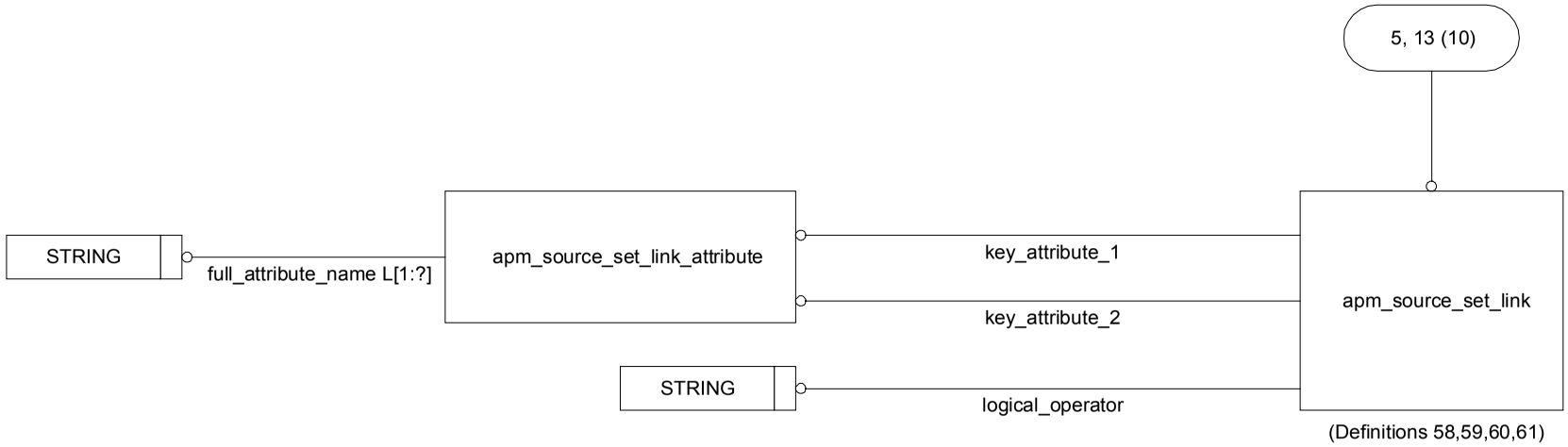


Figure 64-7: APM Source Set Link Entities

APM Relation Entities

Figure 64-8 is an EXPRESS-G diagram showing the entities that form the APM Relation Entities group. Attribute `related_attributes` of entity `apm_relation` corresponds to the list of related primitive attributes in Definition 38-65. However, instead of pointing to a list of APM Primitive Attributes (as specified in the definition) it points to a list of strings that contains only the *names* of the related APM Primitive Attributes.

Abstract entity `apm_relation` is subtyped into `apm_product_relation` and `apm_product_idealization_relation` (defined in Definitions 38-67 and 38-68, respectively).

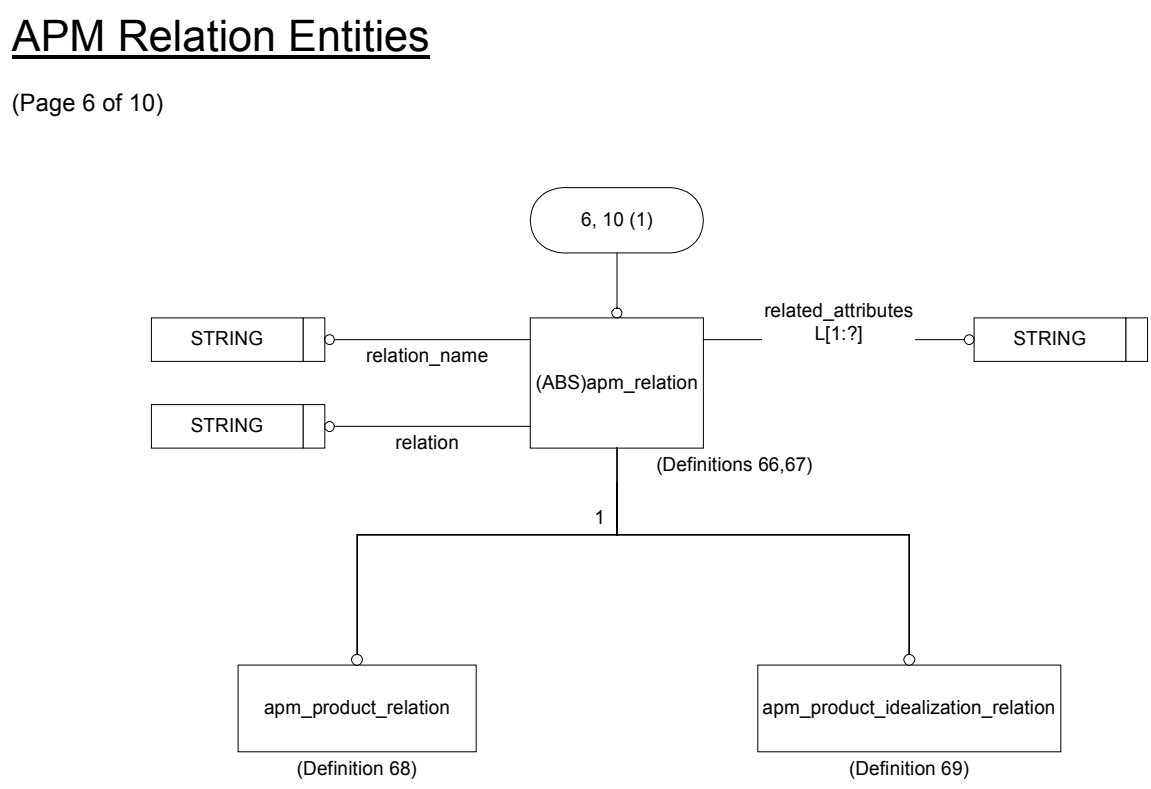


Figure 64-8: APM Relation Entities

Constraint Network Entities

Figure 64-9 shows the EXPRESS-G diagram of the four entities in this group: `constraint_network`, `constraint_network_node`, `constraint_network_relation`, and `constraint_network_variable`.

Entity `constraint_network`, as specified in Definition 38-76, contains a list of constraint network relations (in attribute `relations`) and a list of constraint network variables of the network (in attribute `variables`).

As stated in Definition 38-82, a constraint network node is either a constraint network relation or a constraint network variable. This is implemented in the APM Information Model by subtyping entities `constraint_network_relation` and `constraint_network_variable` from abstract entity `constraint_network_node`. Entity `constraint_network_node` contains two new attributes: `constraint_network` and `marked` (both required). Attribute `constraint_network` points to the constraint network to which the node belongs, and attribute `marked` is a boolean attribute used by the constraint-solving strategy (Subsection 81) to find the nodes connected to a given node.

Entity `constraint_network_relation` contains two new attributes: `active` and `category` (both required). Attribute `active` is a boolean used to indicate whether a relation is active (when its value is `true`) or relaxed (when its value is `false`). Active relations are used to build the systems of equations sent to the constraint solver to try to find the value of an unknown primitive attribute (see Subsection 81). Relaxed relations are just ignored (that is, they do not participate in the systems of equations). Attribute `category` is an integer that may be used to categorize constraint network relations (by assigning the same number to relations of the same category). This attribute is used when APM Relations are mapped into constraint networks to maintain the distinction between product relations and product idealization relations.

Constraint Network Entities

(Page 7 of 10)

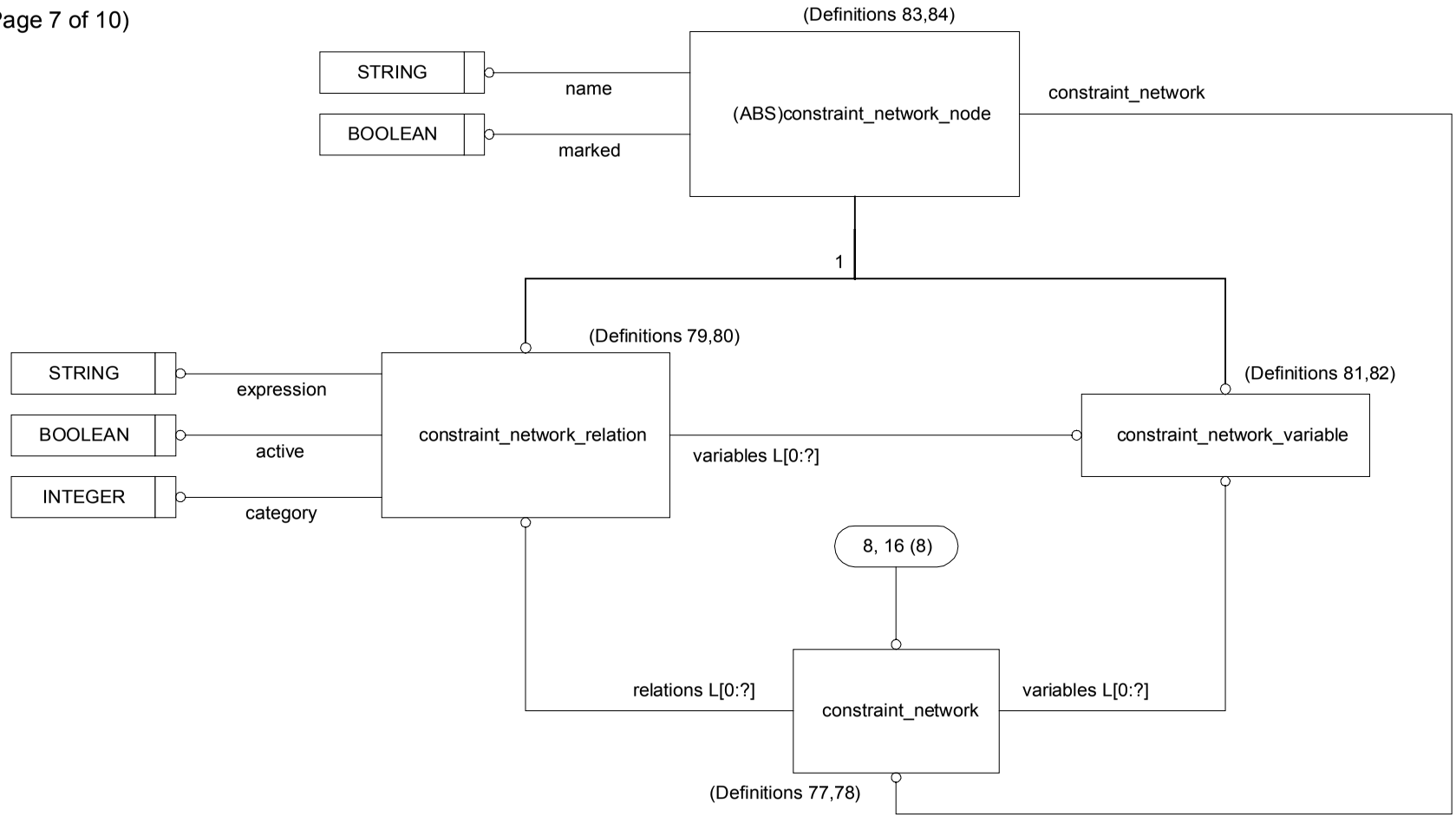


Figure 64-9: Constraint Network Entities

APM Interface Entities

Figure 64-10 is an EXPRESS-G diagram showing the two entities that form the APM Interface Entities group: entity `apm` and the new entity `apm_interface`. Entity `apm` (see Definition 38-69) provides a container for the APM Source Sets, the APM Source Set Links and the Constraint Network of a given APM (attributes `source_sets`, `source_set_links`, and `constraint_network`, respectively). Entity `apm` has two new attributes: `linked_domains` and `linked_instances`. Attribute `linked_domains` is a list of APM Domains whose purpose is to hold the APM Domains that result from linking the individual source sets (see Subsection 78 for details). Similarly, attribute `linked_instances` holds the result of linking the source set instances of each source set (see Subsection 79).

The purpose of entity `apm_interface` is to function as a single point of entry to potentially several APMs involved in a given design-analysis scenario of a given product. Only one global instance of `apm_interface` exists at any given moment, which points to a "current" or "active" APM through its `active_apm` attribute. This global instance of `apm_interface` channels all the operations and requests for information sent to it to the active APM. The `apm_interface` holds a list of APMs (attribute `list_of_apms`), of which only one can be active at any given moment.

APM Source Set Data Wrapper Entities

Figure 64-11 is an EXPRESS-G diagram showing the entities that form the APM Source Set Data Wrapper Entities group. As introduced in Subsection 60, APM Source Set Data Wrapping entities provide a neutral communication mechanism between format-specific data parsers and the APM. Format-specific data wrappers (which are subtyped from `apm_source_data_wrapper_object`) deal with the formatting details of the source set data. They parse the source set data, perform the necessary conversions, and pass it to the APM source set data loading operation in terms of format-independent APM Source Set Data Wrapper Returned Values. As it will be described in Subsection 79, the messages and the information exchanged between the source set data loading operation and the specific data wrapper are independent from the data format being read.

APM Entities

(Page 8 of 10)

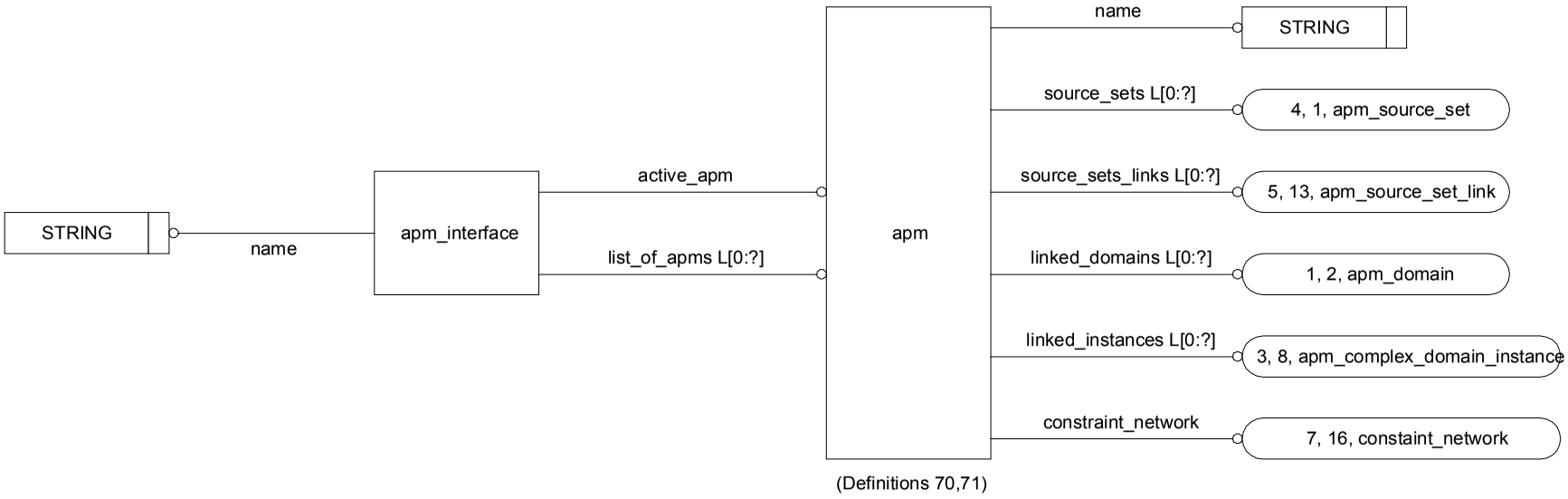


Figure 64-10: APM Interface Entities

The format-specific wrapper returns the information read in terms of one of the subtypes of `apm_source_data_wrapper_returned_value`. Real values are returned as `apm_source_data_wrapper_returned_real_values`, string values as `apm_source_data_wrapper_returned_string_values`, and lists as `apm_source_data_wrapper_returned_lists`. Non-primitive values (objects that have attributes) are returned as `apm_source_data_wrapper_returned_objects`. The APM source set data loading operation knows how to extract the results when the data comes in any of these four forms. More details on how source data wrappers and the source set data loading operation interact will be provided in Subsection 79.

APM Solver Wrapper Entities

Figure 64-12 is an EXPRESS-G diagram showing the three entities that form the APM Solver Wrapper Entities group. As it can be appreciated in the diagram, the entities in this group are very simple (at least structurally - their operations, as it will be described in Subsection 81 are more interesting).

Similarly to the APM Source Set Data Wrapping entities described in the previous subsection, APM Solver Wrapping Entities provide a neutral communication mechanism between the APM get value operations and the particular constraint solver being used. As it will be explained in Subsection 81, mathematical constraint solvers are used to resolve the values of particular APM Instances using the mathematical relations defined among them in the APM.

Wrappers for specific constraint solvers are subtyped from `apm_solver_wrapper`. For example, the diagram in Figure 64-12 shows `mathematica_wrapper` (the constraint solver wrapper implemented in this work) designed to work with the Mathematica system from Wolfram Research (Wolfram 1996). As it will be explained in Subsection 81, where the constraint-solving technique is discussed, APM solver wrappers receive a system of equations from the APM get value operations along with a request for solving for a particular value.

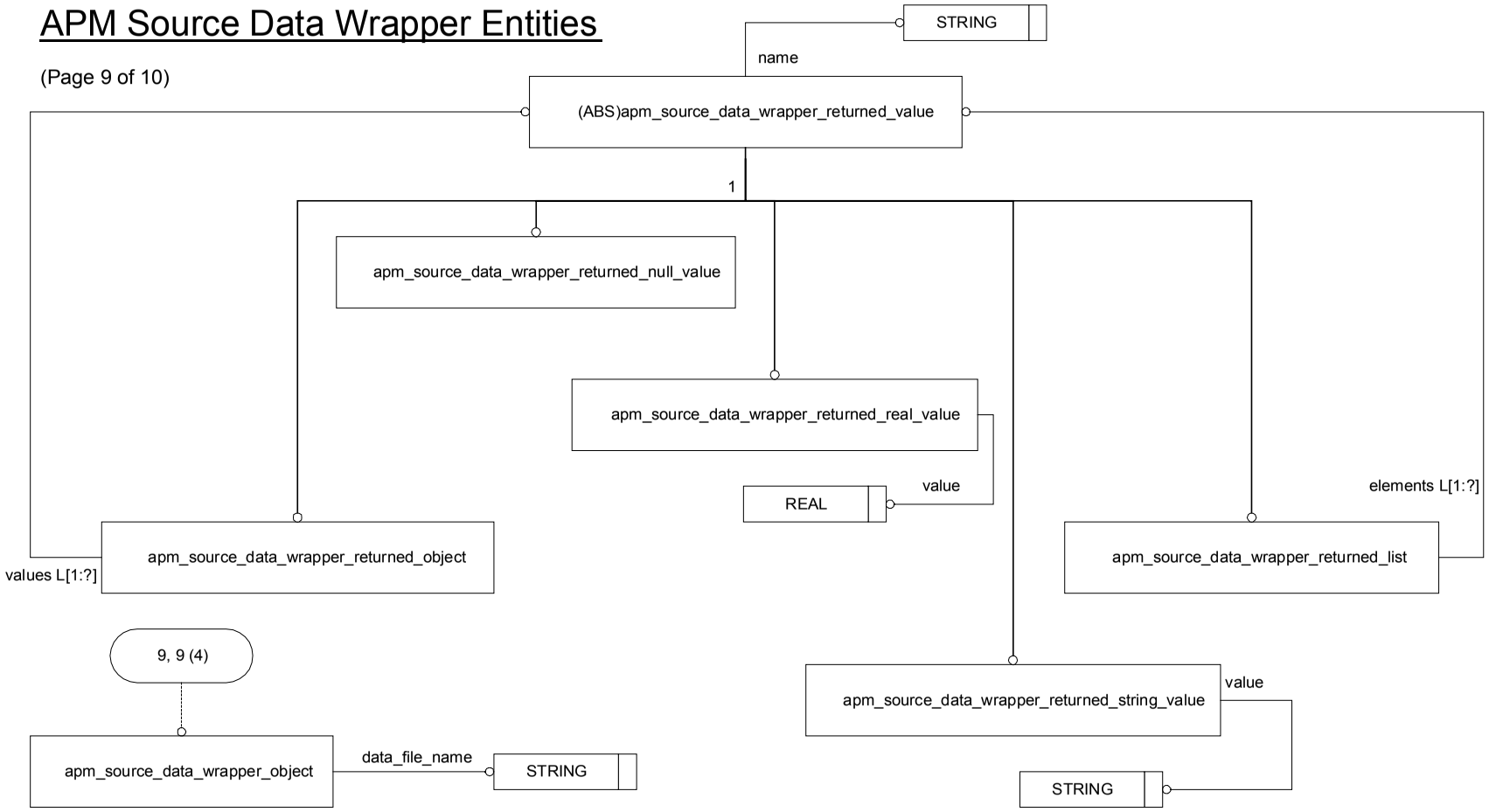


Figure 64-11: APM Source Set Data Wrapper Entities

APM solver wrappers prepare this request for the constraint solver, execute the appropriate solving routines and get the results back from the solver. Finally, they put these results in terms of `apm_solver_results` and send them back to the APM get value operations. As shown in the EXPRESS-G diagram, `apm_solver_results` can only contain a list of real values (in attribute `results`). However, the model could potentially be extended to support other types of values that may be returned by a constraint solver.

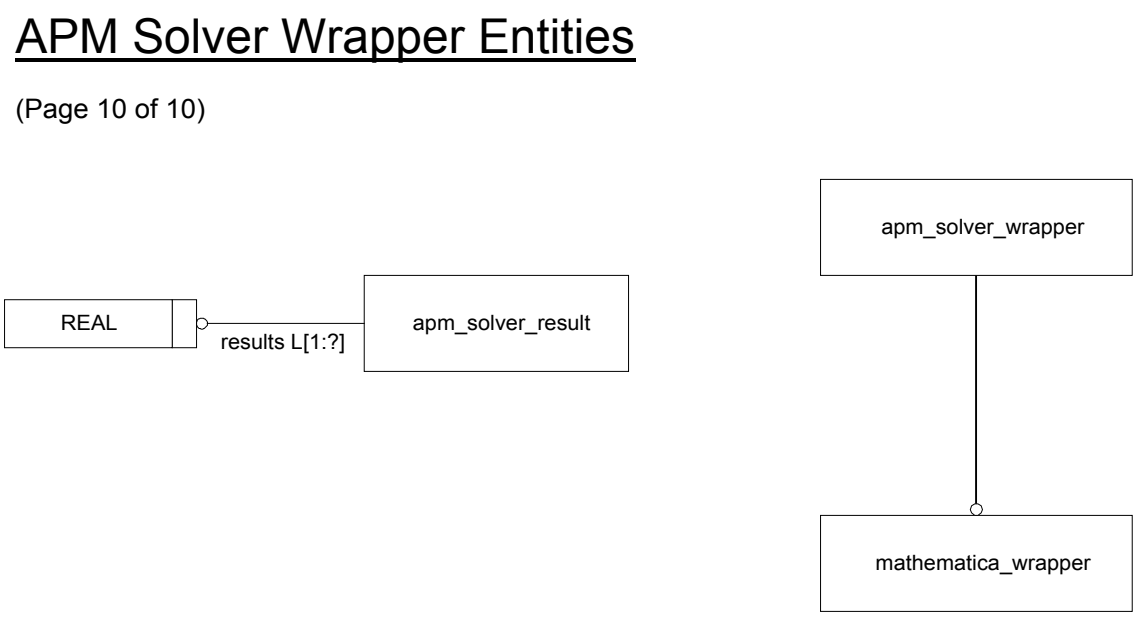


Figure 64-12: APM Solver Wrapper Entities

APM Information Model Implementation in Java

The implementation of the APM Information Model presented in the previous section using the EXPRESS information modeling language is useful to describe the various constructs of the APM in a programming language-independent fashion that is easy to communicate and visualize. However, in order to be able to develop applications that actually use this model, these constructs must also be implemented in some target programming language. As discussed earlier (in Subsection 65), having the APM Information Model described in a

modeling language such as EXPRESS – although not required - greatly facilitates this implementation.

The programming language selected for this work was Sun Microsystems' Java (version 1.1). Java was selected primarily for being a fully object-oriented programming language, as well as for its relative simplicity, portability, robustness, wide availability and popularity (Deitel and Deitel 1998; Deitz 1996; Dragan 1997; Flanagan 1997; Sun Microsystems 1998).

As illustrated in Figure 64-13, the result of implementing the APM Information Model in Java is, in essence, a library of classes (in the case of this work, *packages* of *Java* classes). The operations of the APM Protocol – as it will be presented in Section 77 - will be implemented as *methods* of these classes. Regardless of the language used for implementation, the objective is to provide classes that contain information and functionality to facilitate the development of design-analysis integration applications. Developers can then use these classes to develop APM client applications such as the ones that will be presented in Section 89.

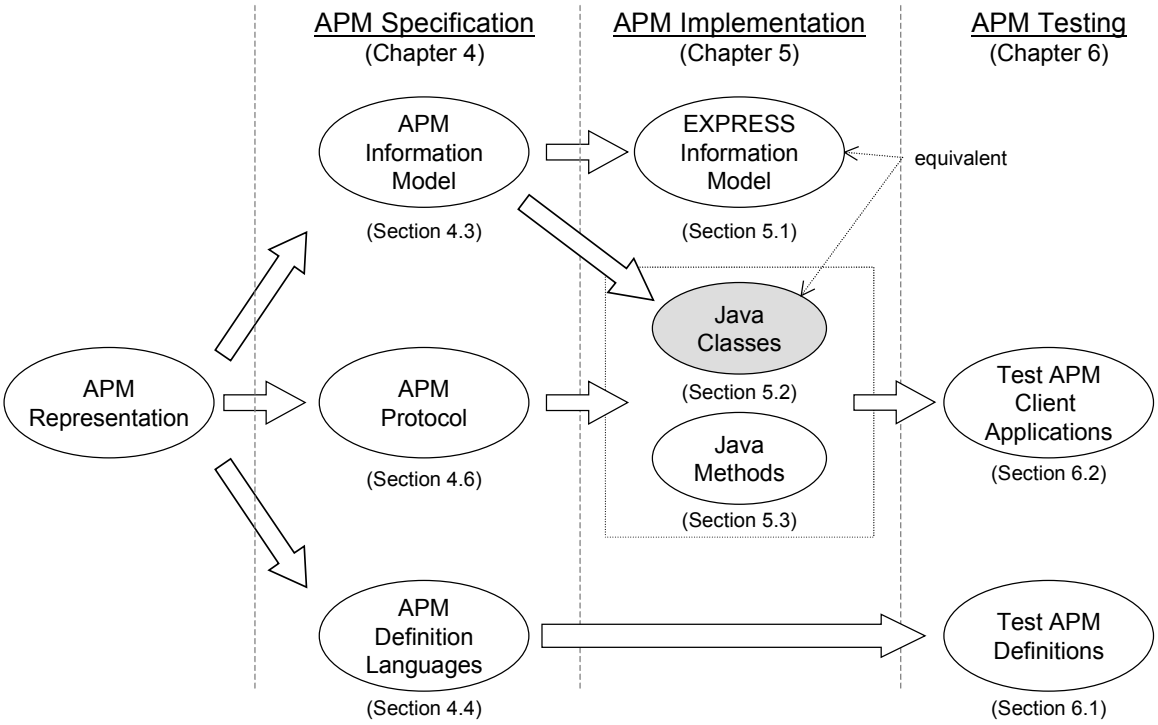


Figure 64-13: APM Information Model Implementation

Figure 64-13 also shows that the EXPRESS implementation of the APM Information Model presented in Section 65 and the Java implementation presented in this section are equivalent. There is virtually a one-to-one correspondence between the entities in the EXPRESS implementation and the classes in the Java implementation: basically, each entity in the EXPRESS APM Information Model was implemented as a Java *class*, and the attributes of these entities became the *class variables* of these classes. For example, entity `apm_object_domain` in the APM Information Model was implemented as the Java class **`APMObjectDomain`**, and attribute `domain_name` of entity `apm_object_domain` became class variable **`domainName`** of the **`APMObjectDomain`** class[42].

The data types defined in the APM Information Model also match the data types in the Java implementation. For example, attribute `domain_name` in the EXPRESS APM Information Model and variable **`domainName`** in the Java implementation are both of type **`String`**. The subtype-supertype hierarchy of the APM Information Model is also replicated in Java. For example, in the APM Information Model entity `apm_object_domain` is a subtype of entity `apm_domain`. Therefore, the Java class **`APMObjectDomain`** is a subtype of class **`APMDomain`**.

For space reasons, the complete description of all the Java classes resulting from this implementation is not included in this thesis. Instead, it was made available on the Internet at (Tamburini 1999), complete with all the attributes, methods and code for each class. A few key classes, however, were included in their entirety in the appendices as a reference to the reader (**`APMInterface`**, **`APM`** and **`APMRealInstance`** in appendices L.1, L.2, and L.3, respectively).

Java classes are normally delivered in logical groups known as *packages*. The packages supplied with this prototype implementation of the APM Information Model and the classes they contain are (▸ indicates inheritance):

1. Package apm

This package contains the core classes of the APM Protocol. These classes are:

- APMDomain

[42] Following the convention used by most object-oriented programmers, class and variable names are written as a single word, with uppercase letters separating individual words. Class names begin with uppercase letters and variable names begin with lowercase letters.

- APMComplexDomain
  - APMObjectDomain
  - APMMultiLevelDomain
- APMPrimitiveDomain
- APMAggregateDomain
  - APMComplexAggregateDomain
  - APMPrimitiveAggregateDomain

- APMAttribute
  - APMComplexAttribute
    - APMObjectAttribute
    - APMMultiLevelAttribute
  - APMPrimitiveAttribute
  - APMAggregateAttribute
    - APMComplexAggregateAttribute
    - APMPrimitiveAggregateAttribute

- APMDomainInstance
  - APMComplexDomainInstance
    - APMObjectDomainInstance
    - APMMultiLevelDomainInstance
  - APMPrimitiveDomainInstance
    - APMRealInstance
    - APMStringInstance
  - APMAggregateDomainInstance
    - APMComplexAggregateDomainInstance
    - APMPrimitiveAggregateDomainInstance

- APMSourceSet
- APMSourceSetLink
- APMSourceSetLinkAttribute
- APMRelation
  - APMProductIdealizationRelation
  - APMProductRelation
- APM
- APMInterface
- ListOfAPMAttributes
- ListOfAPMComplexDomainInstances
- ListOfAPMComplexDomainInstancesPairs
- ListOfAPMComplexDomains
- ListOfAPMDomainInstances
- ListOfAPMDomains
- ListOfAPMObjectDomainInstances
- ListOfAPMObjectDomains
- ListOfAPMPrimitiveDomainInstances
- ListOfAPMRelations
- ListOfAPMSourceSetLinks
- ListOfAPMSourceSets
- ListOfAPMs
- ListOfIntegers
- ListOfReals
- ListOfStrings

2. *Package apm.parser*

This package contains the following two classes:

- APMLexer
- APMParser

These two classes are used to scan and parse APM definition files written in the APM-S definition language (Subsection 52). As discussed in Subsection 78, the **`APMLexer`** class was automatically created using the lexer-generation utility Jlex (Elliot 1997), and the **`APMParser`** class using the parser-generation utility Java-CUP (Hudson 1998). The lexer specification file used to generate **`APMLexer`** using Jlex is included in Appendix E, and the grammar definition file used to generate **`APMParser`** using Java CUP is included in Appendix F.

3. *Package apm.solver*

The classes in this package deal with the wrapping of the constraint solvers and the communication between the APM and the solvers (see Subsection 81). These classes are:

- APMSolverResult
- APMSolverWrapper
  - MathematicaWrapper
- APMSolverWrapperFactory

Class **`MathematicaWrapper`** (a subtype of **`APMSolverWrapper`**) was specifically developed to wrap the constraint-solving system used in this work (Wolfram 1996). This class handles the communication details between the APM and Mathematica. It contains methods to build the constraint-solving request, send it to Mathematica, and interpret the results returned (more details in Subsection 81).

4. Package apm.wrapper

The classes in this package deal with the wrapping of the source design data that is read into the APM (see Subsections 60 and 79 for details). These classes are:

- APMInstanceLexer
- APMInstanceParser
- APMSourceDataWrapperFactory
- APMSourceDataWrapperObject
    - StepWrapper
    - APMInstanceWrapper
- APMSourceDataWrapperReturnedValue
    - APMSourceDataWrapperReturnedObject
    - APMSourceDataWrapperReturnedNullValue
    - APMSourceDataWrapperReturnedRealValue
    - APMSourceDataWrapperReturnedStringValue
    - APMSourceDataWrapperReturnedList
- ListOfAPMSourceDataWrapperReturnedObjects
- ListOfAPMSourceDataWrapperReturnedValues

Class **`StepWrapper`** was developed specifically to read STEP P21 data files. Classes supplied by STEP Tools Inc.'s ST-Developer toolkit (STEP Tools Inc 1997b; STEP Tools Inc 1997c) were used in the development of this class. ST-Developer is a set of software tools for working with EXPRESS information models and EXPRESS-defined data sets. This toolkit provides a library of Java classes for developing STEP applications.

Class **`APMInstanceWrapper`** uses classes **`APMInstanceLexer`** and **`APMInstanceParser`** to scan and parse APM-I files (Subsection 53). The **`APMInstanceLexer`** class was automatically created using the lexer-generation utility

Jlex (Elliot 1997), and the **`APMInstanceParser`** class using the parser-generation utility Java-CUP (Hudson 1998). The lexer specification file used in this work to generate **`APMInstanceLexer`** with Jlex is included in Appendix G, and the grammar definition file used to generate **`APMInstanceParser`** with Java-CUP is included in Appendix H.

5. Package constraint

The classes in this package deal with the creation and manipulation of constraint networks (Subsections 50 and 72) used by the constraint-solving strategy (Subsection 81). These classes are:

- ConstraintNetwork
- ConstraintNetworkNode
  - ConstraintNetworkRelation
  - ConstraintNetworkVariable
- ListOfConstraintNetworkRelations
- ListOfConstraintNetworkVariables

APM Protocol Operations Implementation

Section 58 provided a high-level descriptive specification of the various operations that implementations of the APM Representation should provide. The objective there was to describe *what* the operations should do, instead of *how* they should do it. This section presents a prototype implementation of these operations developed by the author for this work. As illustrated in Figure 64-14, this is the last step of the implementation of the APM Representation.

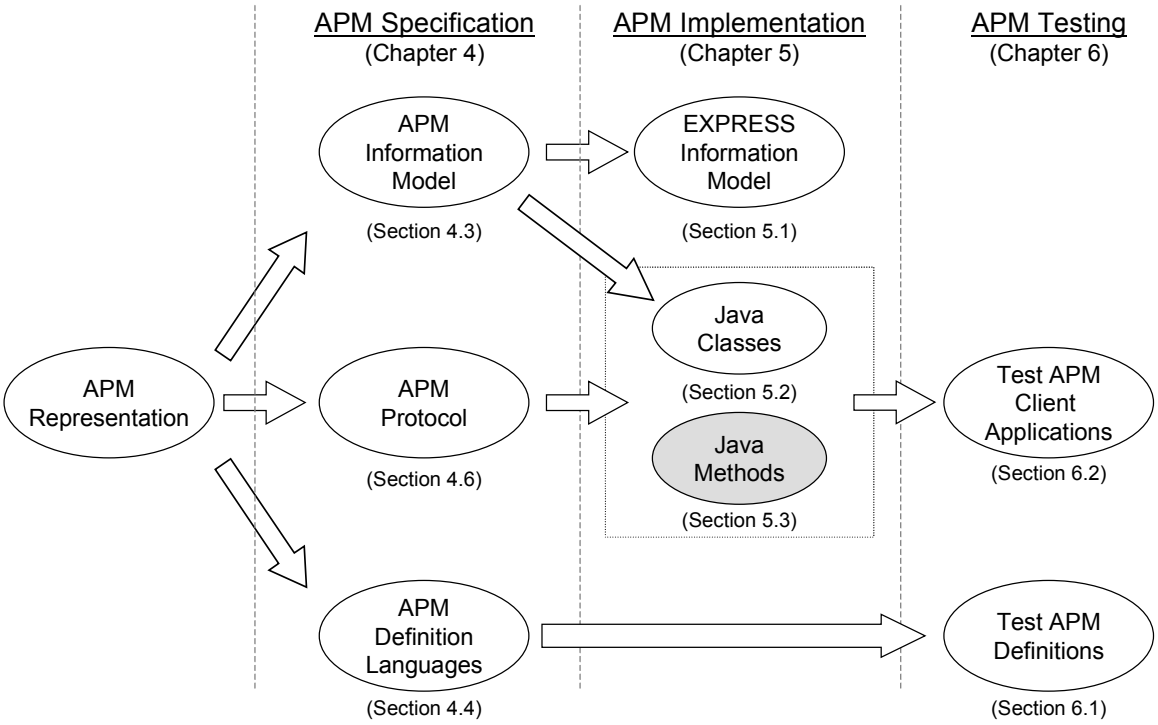


Figure 64-14: APM Protocol Operations Implementation in Java

Essentially, the operations of the APM Protocol were implemented as *class methods* of the Java classes presented in the previous section. In other words, the APM Protocol operations add *behavior* to the APM classes that resulted from the implementation of the APM Information Model.

As discussed in Section 41, the operations defined in the APM Protocol are *late-bound*, that is, they are designed to access and manipulate APM information without previous knowledge of the structure of the domain-specific entities that will be created. In other words, implementations of the APM Representation are not linked to any specific application or domain. This approach allows different client applications to reuse the *same* classes and methods available in the APM class library. More importantly, it also allows the development of generic client applications (such as the APM Browser, described in Subsection 93), designed to work with *any* domain-specific APM.

For space reasons, the complete listing of all the methods resulting from this implementation is not included in this thesis. Instead, it was made available on the Internet at (Tamburini

1999), complete with their source codes. A few key classes, and the complete code of all their methods, were included in the appendices as a reference to the reader (**APMInterface**, **APM** and **APMRealInstance** in appendices L.1, L.2, and L.3, respectively).[43]

Most of the resulting methods are relatively simple and self-explanatory. A large portion of them just retrieve (get) or change (set) the values of the class variables of a given instance. By convention, these methods are named **get<*variable_name*>** and **set< *variable_name*>**. For example, method **getDomainName** of class **APMDomain** gets the value of the variable **domainName** (a string) of an instance of **APMDomain**. Another significant amount of APM methods just query the type of a given instance. By convention, these methods are named **isAn<*class_name*>**. For example, method **isAnAPMObjectDomain** of class **APMDomain** checks if a given instance of **APMDomain** is also an instance of **APMObjectDomain**.

Some of the key operations described in Section 58, however, have more complex purpose and logic. The following subsections will discuss in greater detail how these operations were implemented. When appropriate, their pseudocode and examples of their utilization will be provided. The operations will be grouped by task, in the same way in which they were introduced in Chapter 38.

APM Definitions Loading

The operation to load the APM definitions was implemented as method **loadAPMDefinitions** of class **APMInterface** (from now on, this will be indicated as **APMInterface.loadAPMDefinitions**). The signature of this method is:[44]

```
public static boolean APMInterface.loadAPMDefinitions( String
   apmDefinitionFileName )[45]
```

---

[43] This code should be regarded only as an *example* implementation that illustrates how the various operations of the APM Protocol can be implemented. It is important to keep in mind that this is only a *prototype* implementation, and therefore should not be interpreted as an authoritative specification of how the APM Protocol should be implemented. The assumption is that commercial software developers will develop more robust, efficient and elegant implementations of the APM Protocol than the one provided with this prototype.

[44] The signature defines the name, return type, and arguments of the function. See Appendix Y for more details on the format of a method signature.

[45] APMInterface is a *static* (or global) class, meaning that messages can be sent *to the class* itself, as opposed as to *instances* of the class.

The pseudocode of method **APMInterface.loadAPMDefinitions** is provided in Appendix N, and its source code in Appendix L.1. Essentially, this method creates an instance of class **APM**, sets it as the active APM (the meaning of the active APM is discussed in Subsection 73), and relays the request to load the APM definitions to it. As a result, method **APM.loadAPMDefinitions** gets called. The signature of this method is:

```
boolean APM.loadAPMDefinitions( String apmDefinitionFileName )
```

The pseudocode of method **APM.loadAPMDefinitions** is provided in Appendix O, and its source code in Appendix L.2. This method, in turn, performs the following three actions:

1. Parses the APM Definition File loading the various APM constructs in it define into memory;
2. Links the APM Definitions as specified by the source set links defined in the APM (using method **APM.linkAPMDefinitions**); and
3. Creates the constraint network (using method **APM.createConstraintNetwork**).

In order to parse the APM Definition File, method **APM.loadAPMDefinitions** utilizes the services of the scanning and parsing classes **APMLexer** and **APMParser**. As introduced in Subsection 52, lexical analyzer- and parser-generation utilities are often used in conjunction to create scanning and parsing classes such as **APMLexer** and **APMParser**. Appendix D specifies - in a generic way - the tokens and grammars that define the APM-S language and that can be used as the basis to write utility-specific definition files to generate **APMParser**. The grammar actions are specified in this appendix in pseudocode form. In this thesis, the lexer- and parser-generation utilities used were Jlex (Elliot 1997) and Java CUP (Hudson 1998), respectively. These two tools are very similar to Lex and Yacc, with the difference that they generate Java code instead of C code. The lexical specification used in this work for Jlex is included in Appendix E and the parser specification file used for Java CUP in Appendix F.

The second method called by **APM.loadAPMDefinitions** is **APM.linkAPMDefinitions**. The signature of this method is:

```
void APM.linkAPMDefinitions( )
```

The pseudocode of method **`APM.linkAPMDefinitions`** is provided in Appendix P, and its source code in Appendix L.2. This method links the APM definitions following the procedure outlined in Subsection 46 and illustrated in Subsection 59. Upon completion of method **`APM.linkAPMDefinitions`**, variable **`linkedDomains`** of the active APM contains the linked or "unified" version of the APM structure.

The third and last method performed by **`APM.loadAPMDefinitions`** is **`APM.createConstraintNetwork`**. The signature of this method is:

```
void APM.createConstraintNetwork( )
```

The pseudocode of method **`APM.createConstraintNetwork`** is provided in Appendix Q, and its source code in Appendix L.2. The result of this method is a network of **`ConstraintNetworkRelation`**s and **`ConstraintNetworkVariable`**s connected to each other and stored in variable **`constraintNetwork`** of the active APM.

As shown in Figure 64-15, upon completion of method **`APM.loadAPMDefinitions`**, the active APM has been populated with the following:

1. The APM Source Sets, APM Domains, APM Attributes, APM Relations, and APM Source Set Links defined in the APM Definition File (stored in variable **`sourceSets`**).
2. A linked version of the APM Domains, obtained by linking the domains of the individual source sets according to what is specified by the APM Source Set Links (stored in variable **`linkedDomains`**).
3. A constraint network representing the relations between APM Primitive Attributes in the APM (stored in variable **`constraintNetwork`**).

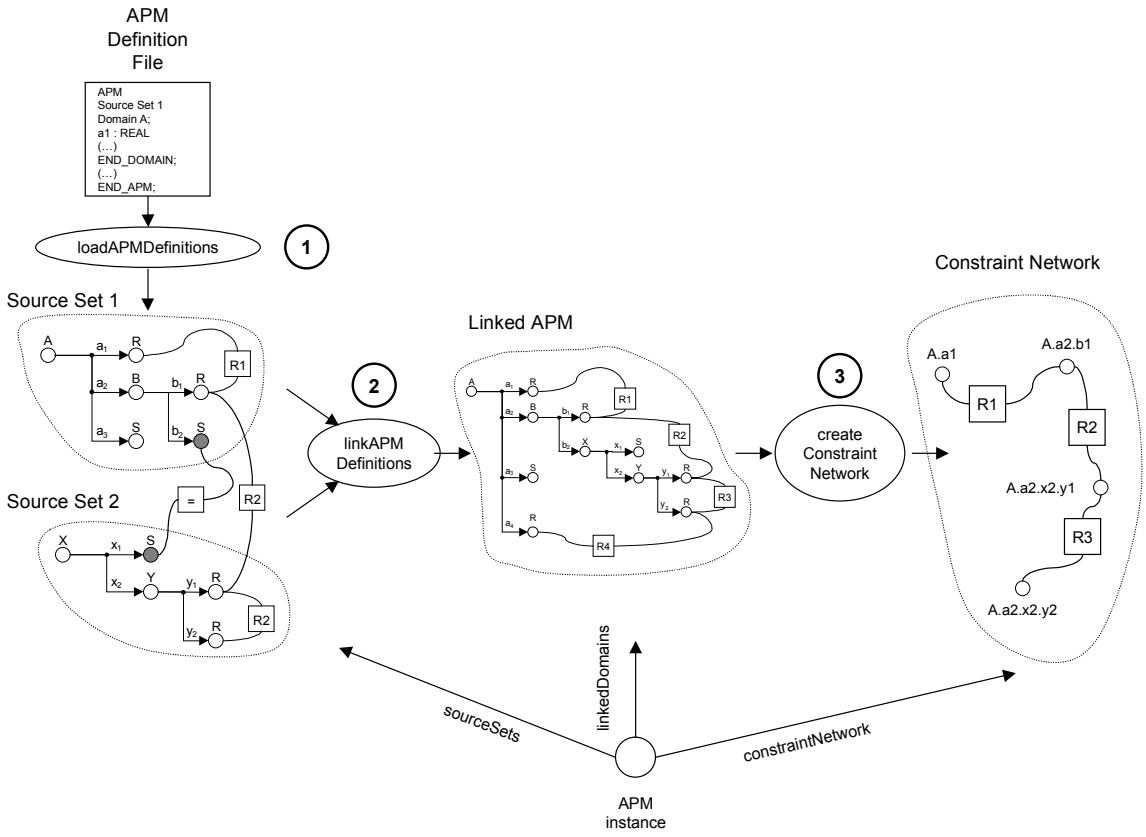


Figure 64-15: APM Definitions Loading Operation

Source Set Data Loading

The operation to load the source set data was implemented as method **`APMInterface.loadSourceSetData`**. The signature of this method is:

```
public static boolean APMInterface.loadSourceSetData( ListOfStrings
   listOfFileNames )
```

The source code of method **`APMInterface.loadSourceSetData`** is provided in Appendix L.1. The list of strings **`listOfFileNames`** contains a list of the file names where the data for each source set is stored. The **`APMInterface`** relays the **`loadSourceSetData`** request to the active APM. As a result, method **`APM.loadSourceSetData`** gets called. The signature of method **`APM.loadSourceSetData`** is:

```
boolean APM.loadSourceSetData( ListOfStrings listOfFileNames )
```

The pseudocode of method **APM.loadSourceSetData** is provided in Appendix S, and its source code in Appendix L.2. This method parses the data stored in the various design repositories and creates the corresponding **APMDomainInstance**s in memory.

Recall from the discussion of Subsection 60 that the proposed approach to load the design data stored in different formats is to use special objects called "source set data wrappers" that parse the design data and transform it into a neutral form understood by method **APM.loadSourceSetData**. These objects are implemented as instances of class **APMSourceDataWrapperObject**. They are created from within method **APM.loadSourceSetData** in the following statement:

```
APMSourceDataWrapperObject wrapperObject =
   APMSourceDataWrapperFactory.makeWrapperObjectFor(
   tempSourceSet.getSourceSetName( ) , tempFileName );
```

In this statement, method **APM.loadSourceSetData** is requesting the **APMSourceDataWrapperFactory** (a static class) to "make" an **APMSourceDataWrapperObject** (**wrapperObject**) for a given source set (**tempSourceSet**) and to connect this wrapper object to a given design repository (**tempFileName**). When the **APMSourceDataWrapperFactory** receives the **makeWrapperObjectFor** message it creates the appropriate wrapper object for the format in question (the pseudocode for method **APMSourceDataWrapperFactory.makeWrapperObjectFor** is provided in Appendix W). The format in which a design repository is stored can be determined by various methods. The method adopted in this implementation is to get the format from a simple database of source set names matched with their respective formats stored in a registry file called **WrapperRegistryFile.txt** such as the one shown in Figure 64-16.

Source Set Name	Format Name
back_plate_geometric_model	Step
back_plate_material_data	Matdb
back_plate_employee_data	SQL
casing_geometric_model	APM-I
casing_material_data	APM-I
flap_link_geometric_model	Step
flap_link_material_data	APM-I
pwa_data	Step
layer_material_data	APM-I
wing_geometry	Step

Figure 64-16: WrapperRegistryFile.txt

Figure 64-17 illustrates the **`APMSourceDataWrapperObject`** creation process. It depicts how the APM sends the request for a wrapper object to the **`APMSourceDataWrapperFactory`** (step 1), how the **`APMSourceDataWrapperFactory`** gets the name of the format from the **`WrapperRegistryFile`** (steps 2 and 3), and finally how the appropriate wrapper object is created and returned to the APM (step 4).

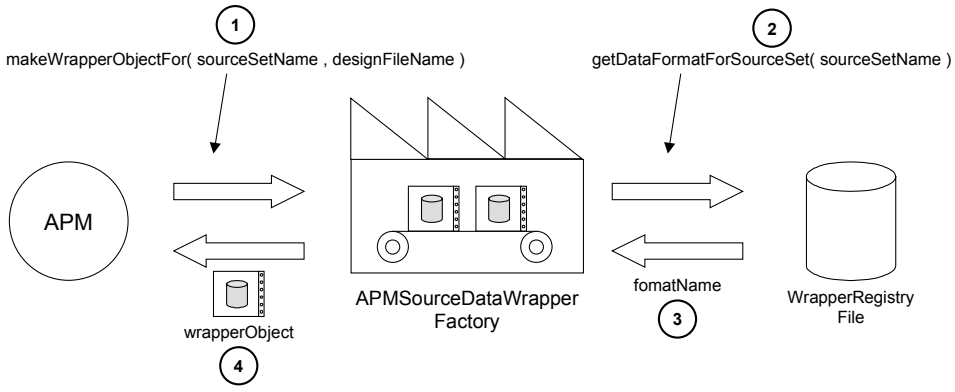


Figure 64-17: APMSourceDataWrapperObject Creation

The **`wrapperObject`** returned by the **`APMSourceDataWrapperFactory`** is used by method **`APM.loadSourceSetData`** to get the values of the source set instances stored in the design repositories. Specific wrappers (such as **`StepWrapper`**) are subtyped from **`APMSourceDataWrapperObject`** and implement method **`APMSourceDataWrapper-`**

Object.getInstancesOf differently (depending on the data format). The APM is not "aware" of (that is, its code is independent from) these differences.

Once the appropriate wrapper object is created, method **APM.loadSourceSetData** issues the following statement:

```
ListOfAPMSourceDataWrapperReturnedObjects returnedListOfObjects =
   wrapperObject.getInstancesOf( tempSourceSetRootDomainSubtype );
```

In this statement, method **APM.loadSourceSetData** is asking **wrapperObject** to parse the data file looking for instances of **tempSourceSetRootDomainSubtype** (which takes the value of each of the subtypes of the root domain). Object **wrapperObject** parses the data file, finds any instances of **tempSourceSetRootDomainSubtype** in it, and returns them in a list of **APMSourceDataWrapperReturnedObjects** (**returnedListOfObjects**). The conversions that take must place in order to build this **returnedListOfObjects** list from the original format of the data are specific to each wrapper.

When method **APM.loadSourceSetData** receives the list of **APMSourceDataWrapper-ReturnedObject**s (**returnedListOfObjects**) from **wrapperObject** it extracts the values from each **APMSourceDataWrapperReturnedObject** in this list and creates corresponding instances of **APMDomainInstances** as follows:

```
instance.populateWithValues( tempReturnedObject.getValues() );
```

where **instance** is an empty instance of **APMComplexDomainInstance** created in a previous step. Method **APMSourceDataWrapperReturnedObject.getValues** extracts the values from **tempReturnedObject** (see the structure of an **APMSourceDataWrapper-ReturnedObject** in Subsection 74) and method **APMComplexDomainInstance.-populateWithValues** fills the **APMComplexDomainInstance instance** with these values.

Notice that this wrapping approach isolates method **APM.loadSourceSetData** from the formatting details of the specific design repositories. These details are handled by the specific data wrappers, which parse the data and package it in a way the APM understands (that is, in terms of **APMSourceDataWrapperReturnedObject**s). The bottom line of the source data wrapping approach is to shift the burden of the formatting details to the data wrappers and keep the communication between the wrappers and method **APM.loadSourceSetData** simple.

Once method **APM.loadSourceSetData** has loaded the data instances from the various design repositories, the next step is to link these instances according to the source set data link definitions specified in the APM. For this purpose, method **APM.loadSourceSetData** calls method **APM.linkSourceSetData**. The signature of method **APM.linkSourceSet-Data** is:

```
void APM.linkSourceSetData( );
```

The pseudocode of method **APM.linkSourceSetData** is provided in Appendix T, and its source code in Appendix L.2. This method works similarly to method **APM.linkAPMDefinitions** described in the previous subsection. The difference is that instead of linking *attributes* as **APM.linkAPMDefinitions** does, **APM.linkSourceSetData** links *instances* of these attributes.

APM Data Usage Operations

1. Retrieving instances of a given domain:

Method **APMInterface.getInstancesOf** can be used to retrieve instances of a given domain in the active APM. Its signature is:

```
public static ListOfAPMComplexDomainInstances
   APMInterface.getInstancesOf( String domainName );
```

For example, referring to the example of Subsection 61, the following statement will return a list of two instances of `flap_link` (one for each flap link stored in the design data file)[46]:

```
ListOfAPMComplexDomainInstances listOfFlapLinkInstances =
   APMInterface.getInstancesOf( "flap_link" );
```

Next, the following statement may be used to get a particular flap link (for example, the second one) from the list returned by **APMInterface.getInstancesOf**:

```
APMComplexDomainInstance flapLinkInstance =
   listOfFlapLinkInstances.elementAt( 1 );
```

[46]The phrase "an instance of flap_link" actually refers to an instance of **APMDomainInstance** (more specifically, in this case, an instance of **APM*Object*DomainInstance**) whose domain name is "flap_link".

2. Getting the value of a primitive attribute:

Methods **`APMRealInstance.getRealValue`** and **`APMStringInstance.getStringValue`** can be used to get the values of an **`APMRealInstance`** or an **`APMStringInstance`**, respectively. The signatures of these two methods are:

```
public double APMRealInstance.getRealValue( )

public String APMStringInstance.getStringValue( )
```

In order to be able to use these two methods, the primitive instance must be obtained first. For this purpose, a combination of methods **`getObjectInstance`**, **`getMultiLevel-Instance`**, **`getRealInstance`** and **`getStringInstance`** of class **`APMComplexDomain-Instance`** can be used to navigate the object-attribute tree in order to get the primitive attribute at its end. The signatures of these methods are:

```
public APMObjectDomainInstance
   APMComplexDomainInstance.getObjectInstance( String attributeName )

public APMMultiLevelDomainInstance
   APMComplexDomainInstance.getMultiLevelInstance( String
   attributeName )

public APMRealInstance APMComplexDomainInstance.getRealInstance(
   String attributeName )

public APMStringInstance APMComplexDomainInstance.getStringInstance(
   String attributeName )
```

For example, the following sequence of methods would be used to get the value **A** of attribute "`area`" in the flap link example of Subsection 61:

```
A = flapLinkInstance.getObjectInstance( "shaft"
   ).getMultiLevelInstance( "critical_cross_section"
   ).getObjectInstance( "simple" ).getRealInstance( "area" ).
   getRealValue( );
```

As indicated in Subsection 61, if the APM Real Instance does not have value, then method **`APMRealInstance.getRealInstance`** triggers a constraint-solving attempt using the relations defined in the APM. The constraint-solving strategy used by **`APMRealInstance.getRealInstance`** will be explained in detail in Subsection 81.

3. Setting the value of a primitive attribute

Methods **APMRealInstance.setValue** and **APMStringInstance.setValue** can be used to set the values of **APMRealInstance**s and **APMStringInstance**s, respectively. Their signatures are:

```
public void APMRealInstance.setRealValue( double value )

public void APMStringInstance.setStringValue( String value )
```

For example, the following statement sets the value of "area" to 15.5:

```
flapLinkInstance.getObjectInstance( "shaft" ).getMultiLevelInstance(
   "critical_cross_section" ).getObjectInstance( "simple"
   ).getRealInstance( "area" ). setValue( 15.5 );
```

4. Accessing APM structure information

Almost all attributes of instances of the APM classes **APM**, **APMSourceSet**, **APMDomain**, **APMAttribute**, **APMRelation**, **APMSourceSetLink**, and **ConstraintNetwork** are accessible through one or more methods defined in the APM Protocol. These methods are of the form **get<*attribute_name*>.** For example, method **getDomainName** of class **APMDomain** gets the value of the variable **domainName** (a string) of an instance of **APMDomain**. A sample of methods from various classes that can be used to access structural APM information is:

Class	Structural APM Information Access Methods
APM	**getSourceSets**, **getSourceSetLinks**, **getConstraintNetwork**
APMSourceSet	**getDomainsInSet**, **getDomain**, **getSubtypesOf**
APMDomain	**getDomainName**
APMComplexDomain	**getLocalRelations**, **getRelations**, **getAttribute**, **getListOfAttributeNames**

APMObjectDomain	**getLocalAttributes**, **getInheritedAttributes**, **getAttributes**, **getSupertypeDomain**, **getLocalAttribute**
APMMultiLevelDomain	**getLevels**, **getLevel**, **getNumberOfLevels**
APMAggregateDomain	**getDomainOfElements**
APMAttribute	**getDomain**, **getAttributeName**, **getContainerDomain**
APMRelation	**getRelationName**, **getRelation**, **getRelatedAttributes**
ConstraintNetwork	**getVariables**, **getRelations**, **getNode**, **getVariable**

APM Constraint-Solving Technique

As mentioned in the previous subsection, when method **getRealValue** is used to request the value of an **APMRealInstance** that does not have a value, a constraint-solving attempt is triggered. A system of equations is built and sent to an external constraint solver, which tries to find a solution and returns any values found. This subsection presents the details of this process; how the constraint network is used to build the system of equations, and how special objects known as APM Solver Wrappers are used to handle the communication of requests and solutions between the APM operations and the constraint solver.

It is important to point out that, in this work, the discussion of constraint solving is limited to *real* values. This is why the focus is on method **APMRealInstance.getRealValue**. Even though it is possible to define constraints between string values, this work will consider only numeric constraints. Therefore, when method **APMStringInstance.getStringValue** is performed and the **APMStringInstance** does not have a value, an error is reported and the method is aborted (that is, no constraint solving is attempted). As mentioned in Subsection 42, other types of **APMPrimitiveDomains** (such as integers) could be added to the APM Information Model in the future, and these primitive domains could have their own

"get<type>Value" methods that also trigger a constraint-solving attempt if they do not have a value.

APMRealInstances have two **boolean** class variables that are of particular interest to this discussion: **hasValue** and **isInput** (Figure 64-5). Variable **hasValue** is set to **true** if the instance has a value (that is, if variable **value** is non-empty) and set to **false** otherwise. Variable **isInput** is set to **true** if the instance is considered an input and set to **false** if it is considered an output. By definition, instances that are inputs must have value, but instances that have value must not necessarily be inputs. To illustrate this, consider the relation used to calculate the effective length of the flap link (from the APM Definition shown in Figure 38-7):

```
"pir1" : effective_length == sleeve_2.center.x - sleeve_1.center.x -
   sleeve_1.radius - sleeve_2.radius
```

In this relation, there are multiple input/output combinations possible. Since it is a linear algebraic equation, any four variables of the relation could be declared as inputs and the fifth would become an output. In order to be able to calculate the value of the output instance, the four instances that were declared as inputs must have value.

In the example above, in which there is only one relation, figuring out the system of equations needed to find the value of an instance is rather trivial. For example, assume that all variables on the right-hand-side of the relation are inputs and have values (say, 25.0, 0.0, 6.5, 7.00, respectively), and that the value of `effective_length` is being solved for. The system of equations for this case would be:

```
effective_length == sleeve_2.center.x - sleeve_1.center.x -
   sleeve_1.radius - sleeve_2.radius

sleeve_2.center.x == 25.0

sleeve_1.center.x == 0.0

sleeve_1.radius == 6.50

sleeve_2.radius == 7.00
```

And the solution:

```
effective_length = 25.0 – 0.0 – 6.50 – 7.00 = 11.50
```

However, if any of the variables in the relation above participates in more than one relation, and these relations, in turn, involve other variables, building the system of equations and figuring out which input/output combinations are valid becomes more complicated. This is where constraint networks (discussed in subsections 50, 72, and 78) come into play. Constraint networks help determine which relations and variables are "connected" to a given variable. Constraint Network Relations and Constraint Network Variables are *nodes* of a Constraint Network. Two nodes of a constraint network are considered to be connected if there is at least one path from one node to the other.

In order to solve for the value of a variable, all the variables and relations connected to it in the constraint network will be used to build the system of equations. To illustrate this approach, consider the constraint network of the flap link example shown in Figure 64-18. As it becomes more evident in the diagram, the values of the different variables even for this simple APM are significantly interconnected. For example, `effective_length` is not only connected to the variables that participate in relation "**pir1**", but also to variables as far in the constraint diagram as the thickness of the flange of the beam (`shaft.critical_cross_section.detailed.tf`).

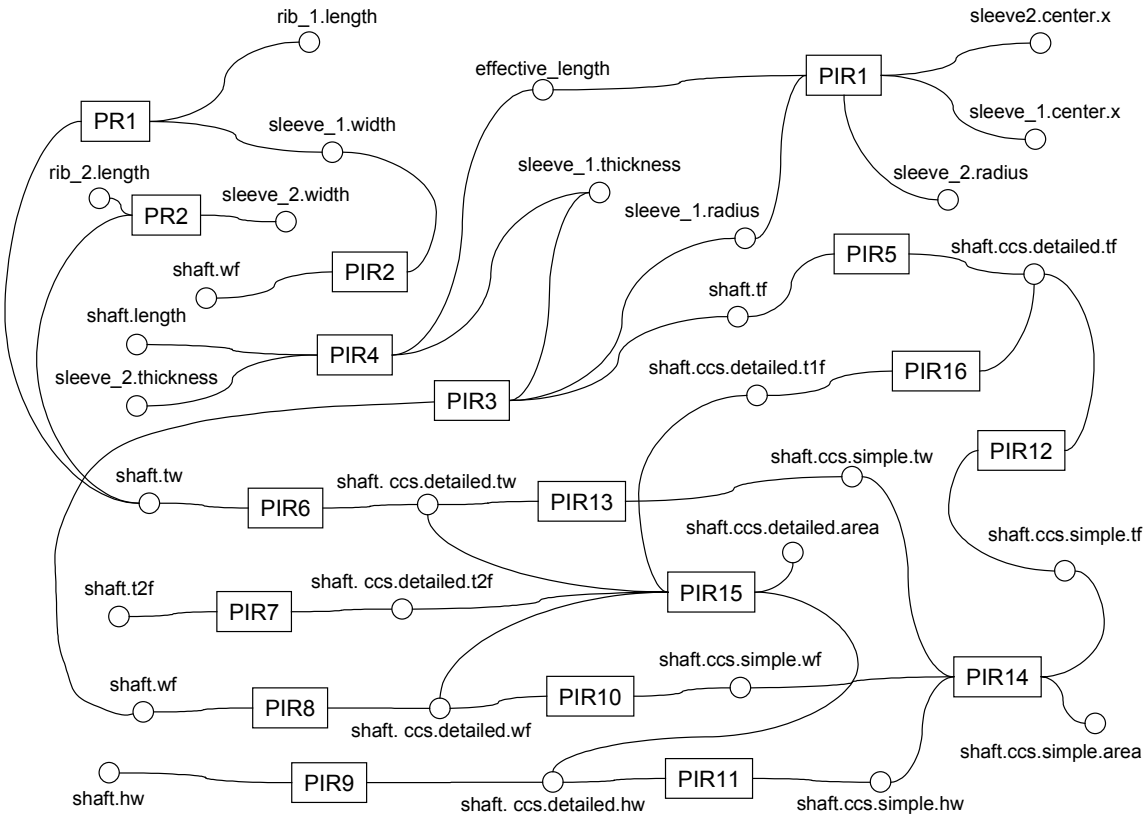


Figure 64-18: Constraint Network for the Flap Link Example

If any of the connected relations is not taken into account when building the system of equations, the values obtained may potentially fall into an inconsistent state. For example, consider the example given earlier where the value of `effective_length` was being solved using only relation "**pir1**", where the value obtained was `effective_length` = 11.50. If the fact that `effective_length` is also connected to relation "**pir4**" is omitted, then "**pir4**" could yield a different result for `effective_length` if the values of `sleeve_1.thickness`, `sleeve_2.thickness`, and `shaft.length` are given. For example, if the following values are given:

```
sleeve_1.thickness = 0.5

sleeve_2.thickness = 0.75

shaft_length = 11.00
```

then relation "**pir4**" :

```
"pir4" : "shaft.length == effective_length - sleeve_1.thickness -
   sleeve_2.thickness
```

would yield a value for `effective_length` of:[47]

```
effective_length = 11.00 + 0.5 + 0.75 = 12.25
```

which is inconsistent with the previously obtained value of 11.50.

By examining the constraint network diagram, it is obvious that when the value of `effective_length` is output from relation "**pir1**" only *two* (and not three) of the remaining variables that participate in relation "**pir4**" (`sleeve_1.thickness`, `sleeve_2.thickness,` `and shaft.length)` can be inputs.

As mentioned previously, the method that triggers the constraint-solving attempt is **`APMRealInstance.getRealValue`**. The pseudocode of method **`APMRealInstance.get-RealValue`** is provided in Appendix U, and its source code in Appendix L.3.

Basically, this method works as follows: if the **`APMRealInstance`** has value, the method simply returns it, otherwise, it will call method **`APMRealInstance.trySolveForValue`**. The signature of this method is:

```
public int APMRealInstance.trySolveForValue()
```

The pseudocode of **`APMRealInstance.trySolveForValue`** is provided in Appendix V, and its source code in Appendix L.3. As the name indicates, this method will attempt to find the value of the instance being requested, by building a system of equations based on the constraint network and sending it to an external constraint solver. If only one solution is found, its value is put in variable **`value`** of the **`APMRealInstance`** and the number **`1`** is returned. If more than one solutions are found, the first positive solution (if any) or the first solution (if they are all negative) is selected, and the number of solutions found is returned.

The task of determining whether or not the system of equations is *solvable* - and finding the solutions to it if it is - is left to an external constraint solver. A variety of reliable and mature constraint solvers are commercially and publicly available (Bjorn and Borning 1992; Borning

[47] The fact that `effective_length` is not on the left-hand-side of the relation does not imply that it cannot be an output. In general, regardless how relations are expressed, *any* variable participating in the relation can be either an input or an output.

and Freeman-Benson 1998; Borning, Marriott et al. 1997; Leler 1988; Marriott and Stuckey 1998), and therefore it makes more sense to use them as external constraint-solving engines than replicating their codes inside APM methods.

To illustrate the **`APMRealInstance.trySolveForValue`** method, consider the simplified version of the constraint network of Figure 64-18 shown in Figure 64-19 (taking only **"pir1"** and **"pir4"** into account). The figure indicates with labels which variables are inputs and which outputs, and the values of the inputs (as loaded from the design repositories).

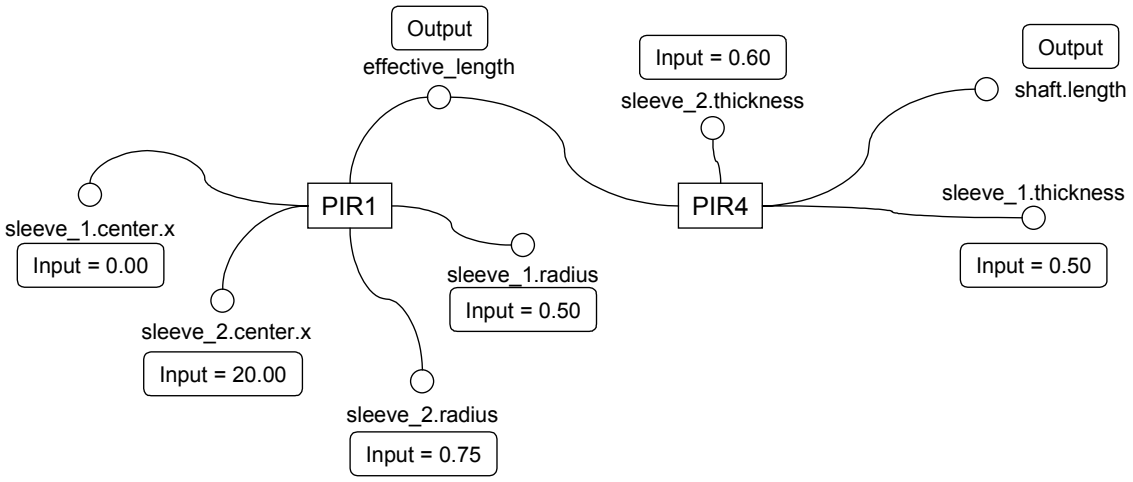


Figure 64-19: Constraint Network for the Flap Link Example Indicating Inputs and Outputs

Assume now that the value of `shaft.length` has been requested with the following statement:

```
L = flapLinkInstance.getObjectInstance( "shaft" ).getRealInstance(
   "length" ).getRealValue( );
```

(where **`flapLinkInstance`** is an instance of flap link found using method **`APMInterface.getInstancesOf`**, as described in Subsection 80)

Since `length` does not yet have a value, method **`APMRealInstance.trySolveForValue`** will be triggered. The relations connected to this instance are "**pir4**" and "**pir1**", and the connected variables are `sleeve_1.thickness`, `sleeve_2.thickness`, `effective_length`, `sleeve_1.radius`, `sleeve_2.radius`, `sleeve_1.center.x`, and `sleeve_2.-`

`center.x`. Of these connected variables, all except `effective_length` are inputs and have value. Thus, the following system is sent to the constraint solver:

Solve for:

```
shaft.length
```

Given:

```
effective_length = sleeve_2.center.x - sleeve_1.center.x -
   sleeve_1.radius - sleeve_2.radius ("pir1")

shaft.length = effective_length - sleeve_1.thickness -
   sleeve_2.thickness ("pir4")

sleeve_1.thickness = 0.5

sleeve_2.thickness = 0.6

sleeve_1.radius = 0.5

sleeve_2.radius = 0.75

sleeve_1.center.x = 0.0

sleeve_2.center.x = 20.00
```

The solver will return the following result:[48]

```
shaft.length = 17.65
```

Method **`APMRealInstance.trySolveForValue`** uses the services of special objects – called "solver wrapper objects" – to handle the communication with the external constraint solver. First, it creates an instance of **`APMSolverWrapper`** as follows:

```
APMSolverWrapper solver =
   APMSolverWrapperFactory.makeSolverWrapperFor( "mathematica" );
```

`APMSolverWrapperFactory` is a static class whose only method is **`makeSolverWrapperFor`**. Its task is to create a new instance of the appropriate subtype of

[48] Notice that, in the process of finding a solution for `shaft.length`, a solution for `effective_length` was also found as a by-product. In the current implementation, by-product solutions such as this are not stored anywhere for potential future utilization. As a result, if `effective_length` is requested later, it will be calculated with its own request to the constraint solver. A recommended extension – stated in Chapter 110 - is to take advantage of by-product solutions to improve performance.

APMSolverWrapper (**MathematicaWrapper**, in this case) depending on the solver indicated by the argument ("**mathematica**").

Class **APMSolverWrapper** has method **solveFor**, whose signature is:

```
public APMSolverResult APMSolverWrapper.solveFor( String variableName
   , ListOfStrings relations , ListOfStrings inputVariableNames ,
   ListOfReals inputValues )
```

The arguments passed to **APMSolverWrapper.solveFor** define the system of equations to be solved by the constraint solver. Argument **variableName** is the name of the variable that is being solved for, **relations** contains the list of equations, **inputVariableNames** is a list of names of the variables that have value (that is, that are input variables), and **inputValues** are their corresponding values.

As a result of this method, the **APMSolverWrapper** instance returns an instance of **APM-SolverResult**, which contains a list of real numbers corresponding to the solution(s) found (see Subsection 75). This list of solutions is extracted from the **APMSolverResult** instance with method **APMSolverResult.getResults()** (which returns a **ListOfReals**). In addition, method **APMSolverResult.hasResults()** can be used to find out whether or not the **APMSolverResult** instance contains any result.

The **APMSolverWrapper** instance handles the solver-specific details of building the system of equations, sending the appropriate commands to the constraint solver and interpreting the results for the APM. For example, When method **solveFor** is performed on **MathematicaWrapper**, it creates the following sequence of Mathematica commands (Wolfram 1996):

```
output = OpenWrite["mathematica_result.txt");
solutions = ReplaceList[ variable , Cases[ Union[ ToRules[ Reduce[
   equations , variable ]] ] , Rule[ variable , y_ ] /; NumberQ[y] ]
   ];
WriteString[ output , Map[ CForm , solutions ] ];
Close[output];
Exit[];
```

This instructs Mathematica to solve for **variable** given the system of equations contained in **equations**, and return the solutions in a file called **"mathematica_result.txt"** in the form:

{ solution$_1$, solution$_2$, … , solution$_n$ }

MathematicaWrapper then parses this result and builds the corresponding **APMWrapperResult** instance, which is finally returned to the calling method **APMRealInstance.trySolveForValue**.

When the design data is loaded into the APM, all the attributes that have value in the design repositories are initially set as inputs and all the remaining attributes as outputs. Depending on the nature of the client application that is using the APM data, however, it could make sense to allow the user to override these initial assignments at run time, effectively changing the input/output combinations of the variables in the constraint network. Consider, for example, the process of refining a preliminary design, in which:

1. Preliminary values are initially assigned to design attributes of the part.
2. These preliminary design values are used to calculate other derived or idealized values.
3. Analysis is performed using a combination of preliminary design, derived and idealized values.
4. Refined values for some derived and idealized attributes are obtained as a result of analysis.
5. Refined design values are calculated from these refined derived and idealized values.

Steps 4 and 5, in which analysis results are used to determine the value of design attributes, are typical of design *synthesis* (Section 5). In the last step, the input/output combinations that were used to calculate the derived and idealized values from the preliminary design values in step 2 change; an idealized attribute that was an output in step 2, became an input in step 5.

In order to enable this run-time definition of input/output combinations, class **APMRealInstance** provides methods **setAsInput** and **setAsOutput**, that can be used to declare an **APMRealInstance** as an input or as an output, respectively. The signatures of these methods are:

```
public void APMRealInstance.setAsInput( )

public void APMRealInstance.setAsOutput( )
```

Together, these two methods allow the development of applications in which the user can dynamically switch between analysis and synthesis scenarios. Subsection 92 presents an APM application that uses these two methods for such a purpose.

Besides simply toggling the value of **APMRealInstance** variable **isInput**, these two methods must take into account the effect that changing an instance from input to output (or vice versa) has on the rest of the instances in the constraint network. When an instance is set as an output with method **APMRealInstance.setAsOutput**, the following occurs:

1. The values of all instances connected to this instance in the constraint network that are also outputs are reset (that is, the value of their **hasValue** variable is set to **false**) so that they are recalculated the next time their values are queried:

   ```
   this.resetConnectedOutputs();
   ```

2. The value of variable **isInput** is set to **false**:

   ```
   this.isInput ← false;
   ```

3. The value of variable **hasValue** is set to **false** (this forces the calculation of the value of this instance next time is queried):

   ```
   this.hasValue ← false;
   ```

When method **APMRealInstance.setAsInput** is performed, the only action performed is setting the value of variable **isInput** to **true**:

```
this.isInput ← true;
```

Method **APMRealInstance.setValue** (introduced in Subsection 80) also has an effect on the other instances connected in the constraint network. When this method is performed, the following occurs (this method takes a real argument **v**):

1. Set the value of variable **value** to **v**:

   ```
   this.value ← v;
   ```

2. Set the value of variable **hasValue** to true

   ```
   this.hasValue ← true;
   ```

3. If this instance is an input and if it has value, reset the values of the connected instances that are outputs (because they need to be recalculated the next time their values are queried):

```
If this.isInput() AND this.hasValue()
   this.resetConnectedOutputs();
```

The APM Protocol also provides a method for inactivating or "relaxing" a relation in the constraint network (method **`ConstraintNetworkRelation.setActive`**). When method **`ConstraintNetworkRelation.setActive( false )`** is used to set the value of variable **`isActive`** of a relation from **`true`** to **`false`**, it effectively removes the relation from the constraint network. When a relation is inactive, it is not taken into account by method **`ConstraintNetworkNode.getConnectedRelationsExpressions`**, which is used within **`APMRealInstance.trySolveForValue`** to build the system of equations to be sent to the constraint solver.

Subsection 92 demonstrates the utilization of method **`ConstraintNetworkRelation.set-Active`** with an APM application that allows the user to activate or deactivate relations, effectively changing the topology of the constraint network also at run time.

Finally, it should be pointed out that the behavior specified by the APM Protocol for methods **`APMRealInstance.setAsInput`** and **`APMRealInstance.setAsOutput`** is rather naïve, since the responsibility of determining which input/output combinations are valid for a given constraint network is left entirely to the programmer or to the user. As a consequence, it is possible to specify invalid input/output combinations that lead to conflicting solutions or to no solutions at all. For example, consider the simple constraint network of Figure 64-20 conformed by two linear relations (**a + b + c = 0** and **c + x – y = 2)** and five variables (**a**, **b**, **c**, **x** and **y**). A valid input/output combination for this example would be:

Inputs: **`a`**, **`b`**, **`x`**

Outputs: **`c`**, **`y`**

But there is nothing to prevent the user from specifying the following combination:

Inputs: **`a`**, **`b`**, **`x`**, **`y`**

Outputs: **c**

which could lead to a conflicting result (because two different values for **c** could be obtained; one with **a** and **b** and another with **x** and **y**).

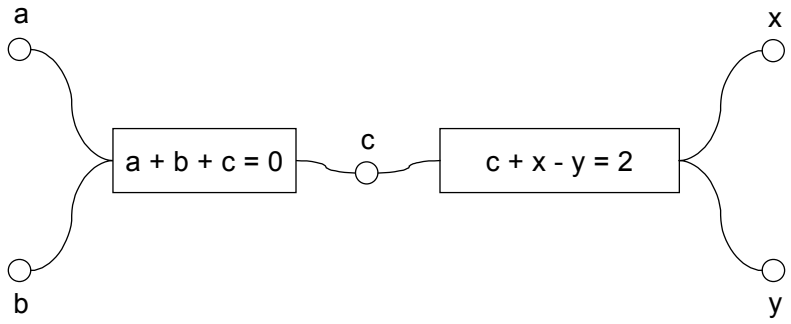


Figure 64-20: Constraint Network Input/Output Combinations

A more sophisticated version of method **APMRealInstance.setAsInput** could inspect the constraint network and automatically flag other variables as outputs as the user declares the inputs. In the example above, if the user sets **a** and **b** as inputs, the method could automatically inspect the constraint network and determine that **c** is an output. Conversely, method **APMRealInstance.setAsOutput** could determine which variables should be given as inputs if the user wants a given variable as an output. For example, if the user wants to obtain **c** as an output, the method could suggest **a** and **b** or **x** and **y** as inputs. This functionality may be quite complex to implement (particularly if the relations in the constraint network are not linear or algebraic as the ones shown in the example) and, for the purposes of this work, it will be considered out of scope.

APM Data Saving Operations

Class **APMInterface** provides two methods to save the linked and unlinked APM instances as specified in Subsection 0, respectively:

```
public static void APMInterface.saveLinkedInstances(String
   outputFileName)

public static void APMInterface.saveInstancesBySourceSet(
   ListOfStrings outputFileNames )
```

These two methods relay the request to the active APM by calling the following two methods:

```
void APM.saveLinkedInstances(String outputFileName)

void APM.saveInstancesBySourceSet(ListOfStrings outputFileNames)
```

The first method saves the instances in file **outputFileName** conforming to the linked version of the APM. The second method unlinks these instances and saves them in separate files, one for each source set defined in the APM.

CHAPTER 6

TEST CASES

This chapter presents a series of test cases developed by the author that test and validate the APM concepts introduced in this thesis so far. As illustrated in Figure 83-1, these test cases include Test APM Definitions and Test APM Applications. Section 84 presents four APM Definitions that use the APM-S Definition Language introduced in Chapter 38. Section 89 presents four APM Applications that demonstrate the utilization of the Java classes and methods presented in Chapter 64. As also shown in the figure, the Test APM Applications presented in Section 89 use the APM Definitions of Section 84.

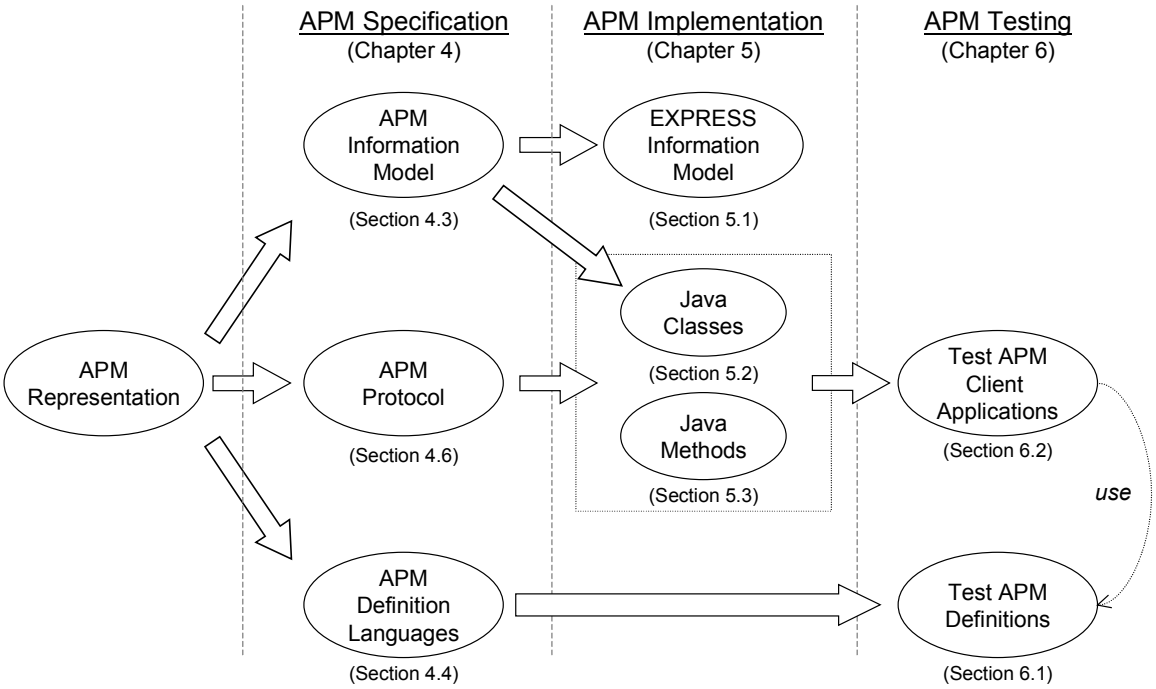


Figure 83-1: APM Representation Testing

Finally, Section 94 presents two test cases demonstrating a couple of strategies for interfacing the APM approach with commercial solid modeling systems.

The results of these test cases will help evaluate – in Chapter 97 – how well the APM Representation meets the objectives set in Chapter 27 and how good this representation is at facilitating design-analysis integration.

Test APM Definitions

The test APM definitions presented in this section are, in essence, APM-S definitions for specific types of parts or products. These APM definitions define analyzable views of these parts or products that can be used by APM client applications for different analysis purposes. For example, the APM Browser (one of the test APM client applications developed for this thesis - presented in Subsection 93) can read the Flap Link APM (one of the test APM definitions - presented in Subsection 85) - along with its corresponding design data - in order to display the structure of the APM and the values of the various attributes defined in it. Alternatively, the Flap Link Extensional Analysis Application (another test APM client application - presented in Subsection 91) can read the *same* APM definition to perform an elongation analysis of a flap link.

Associated with each test APM definition, there is also a collection of one or more repositories of design data (also known as "source data repositories") that provide the data instances needed to run the analyses and obtain results that can be evaluated later. In this work, both APM-I and STEP P21 are used interchangeably as the formats for these repositories of design data.

The following four test APM definitions – each corresponding to a different engineering part - will be presented in the subsections that follow:

1. Flap Link APM (Subsection 85);
2. Back Plate APM (Subsection 86);

3. Airplane Wing Flap Support APM (Subsection 87)[49]; and
4. Printed Wiring Assembly APM (Subsection 88).

Flap Link APM

The Flap Link APM was briefly introduced in the previous chapter and used in several occasions to illustrate some of the APM concepts defined there. This subsection will provide a more detailed discussion of this APM.

The flap link is a fictitious part assumed to be a component of an airplane wing flap mechanism assembly. It is simply a rod that connects two parallel shafts, and assumed to be loaded only in tension. As shown in Figure 83-2, the flap link is composed of two sleeves (sleeve 1 and sleeve 2), a shaft, and two ribs (rib 1 and rib 2). The shaft that connects the two sleeves has an I-shaped cross section (Figure 83-3) of variable height and width (h_w and w_f, respectively).

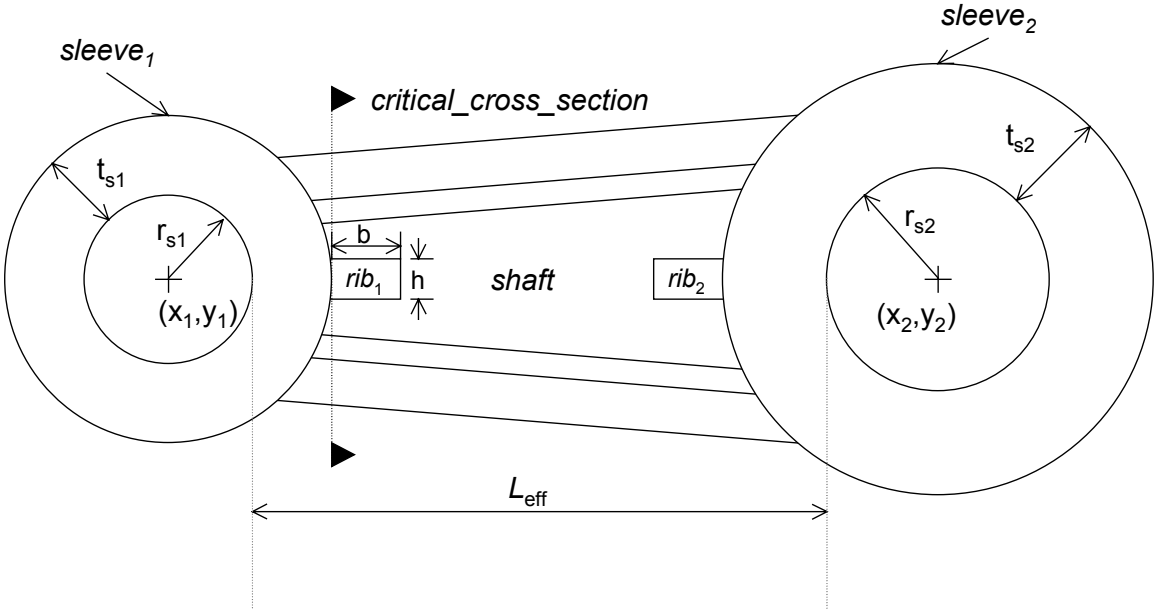


Figure 83-2: Airplane Wing Flap Linkage ("Flap Link")

[49] This APM definition differs from the others in that it defines an APM for a one-of-a-kind part. The other three APMs can potentially be used for *families* of part instances.

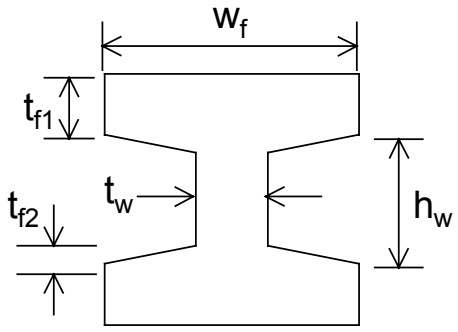


Figure 83-3: Flap Link Cross Section

The APM definition of the flap link is shown in Figures 83-4 and 83-5. This APM definition contains two source sets: `flap_link_geometric_model` and `flap_link_material_-properties`. The first source set (`flap_link_geometric_model`) contains domains to define the geometry and features of the flap link. The second source set (`flap_link_material_properties`) contains domains to define materials and their mechanical properties. There is only one source set link between these two source sets, which links attribute `material` of instances of domain `flap_link` from the first source set with instances of domain `material` from the second source set when the value of `flap_link.material` (a string) is equal to the value of `material.name`.

```
APM flap_link;

SOURCE_SET flap_link_geometric_model ROOT_DOMAIN flap_link;

DOMAIN flap_link;
  ESSENTIAL part_number : STRING;
  IDEALIZED effective_length : REAL;
  sleeve_1 : sleeve;
  sleeve_2 : sleeve;
  shaft : beam;
  rib_1 : rib;
  rib_2 : rib;
  ESSENTIAL material : STRING;
 PRODUCT_RELATIONS
  pr1 : "<rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2";
  pr2 : "<rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2";
 PRODUCT_IDEALIZATION_RELATIONS
  pir1 : "<effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> - <sleeve_1.radius> -
          <sleeve_2.radius>";
  pir2 : "<shaft.wf> == <sleeve_1.width>";
  pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> - <shaft.tf> )";
  pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness> - <sleeve_2.thickness>";
END_DOMAIN;

DOMAIN sleeve;
  ESSENTIAL width : REAL;
  ESSENTIAL thickness : REAL;
  ESSENTIAL radius : REAL;
  center : coordinates;
END_DOMAIN;

DOMAIN coordinates;
  ESSENTIAL x : REAL;
  ESSENTIAL y : REAL;
END_DOMAIN;

DOMAIN beam;
  critical_cross_section : MULTI_LEVEL cross_section;
  length : REAL;
  ESSENTIAL tf : REAL;
  ESSENTIAL tw : REAL;
  ESSENTIAL t2f : REAL;
  ESSENTIAL wf : REAL;
  ESSENTIAL hw : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir5 : "<critical_cross_section.detailed.tf> == <tf>";
  pir6 : "<critical_cross_section.detailed.tw> == <tw>";
  pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
  pir8 : "<critical_cross_section.detailed.wf> == <wf>";
  pir9 : "<critical_cross_section.detailed.hw> == <hw>";
END_DOMAIN;

MULTI_LEVEL_DOMAIN cross_section;
  detailed : detailed_I_section;
  simple : simple_I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir10 : "<detailed.wf> == <simple.wf>";
  pir11 : "<detailed.hw> == <simple.hw>";
  pir12 : "<detailed.tf> == <simple.tf>";
  pir13 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;

DOMAIN simple_I_section SUBTYPE_OF I_section;
 PRODUCT_IDEALIZATION_RELATIONS
  pir14: "<area> == 2*<wf>*<tf> + <tw>*<hw>";
END_DOMAIN;

DOMAIN detailed_I_section SUBTYPE_OF I_section;
  IDEALIZED t1f : REAL;
  IDEALIZED t2f : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
  pir15: "<area> == <wf>*( <t1f> + <t2f> ) + <tw>*( <t2f> - <t1f> ) +
          <tw>*<hw>";
  pir16: "<t1f> == <tf>";
END_DOMAIN;

DOMAIN I_section;
  IDEALIZED wf : REAL;
  IDEALIZED tf : REAL;
  IDEALIZED tw : REAL;
  IDEALIZED hw : REAL;
  IDEALIZED area : REAL;
END_DOMAIN;

DOMAIN rib;
  ESSENTIAL base : REAL;
  ESSENTIAL height : REAL;
  length : REAL;
END_DOMAIN;

END_SOURCE_SET;
```

Figure 83-4: Flap Link APM Definition

```
SOURCE_SET flap_link_material_properties ROOT_DOMAIN material;

DOMAIN material;
  ESSENTIAL name : STRING;
  stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN material_levels;
  temperature_independent_linear_elastic : linear_elastic_model;
  temperature_dependent_linear_elastic : temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;

DOMAIN linear_elastic_model;
  IDEALIZED youngs_modulus : REAL;
  IDEALIZED poissons_ratio : REAL;
  IDEALIZED cte : REAL;
END_DOMAIN;

DOMAIN temperature_dependent_linear_elastic_model;
  IDEALIZED transition_temperature : REAL;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
  flap_link_geometric_model.flap_link.material == flap_link_material_properties.material.name;
END_LINK_DEFINITIONS;

END_APM;
```

Figure 83-5: Flap Link APM Definition (continued)

Domain flap_link is the root domain of the first source set of this APM (recall the meaning of a root domain in Subsection 69). This domain contains eight attributes: part_number (the part number of the flap link, of type STRING), effective_length (the effective length of the flap link, an idealized attribute of type REAL), sleeve_1 and sleeve_2 (the two sleeves of the flap link, of type sleeve), shaft (the beam that connects the two sleeves, of type beam), rib_1 and rib_2 (the two ribs, of type rib), and material (the name of the material of which the flap link is made, of type STRING).

Six relations are defined in domain flap_link. Two of these relations (pr1 and pr2) are *product relations* and four (pir1 through pir4) are *product idealization relations* (see Subsection 48). Product relations pr1 and pr2 are defined as follows:

```
pr1 : "<rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2";
pr2 : "<rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2";
```

These two relations relate the lengths of ribs 1 and 2 with the widths of sleeves 1 and 2, respectively, and the thickness of the web of the shaft. These relations are considered *product relations* because they are directly derived from the geometry of the flap link. On the other hand, relations pir1 through pir4:

```
pir1 : "<effective_length> == <sleeve_2.center.x> -
        <sleeve_1.center.x> - <sleeve_1.radius> - <sleeve_2.radius>";
pir2 : "<shaft.wf> == <sleeve_1.width>";
pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> -
        <shaft.tf> )";
pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness> -
        <sleeve_2.thickness>";
```

are considered *product idealization relations* because they are heuristic in nature (they are derived from some rule of thumb or arbitrary simplification), or because they define how idealized attributes relate with other product or idealized attributes. For example, relation pir2 states that the width of the flange of the shaft (wf) is always equal to the width of sleeve 1 (thus ignoring the fact that wf is actually variable). Relation pir3 assumes that the height of the web of the shaft (hw) is also a constant equal to the radius of sleeve 1 plus the thickness of sleeve 1 minus the thickness of the flange of the shaft multiplied by two. Relation pir1 defines how the idealized attribute effective_length is related to the centers and the radiuses of the sleeves.

The Flap Link APM demonstrates the utilization of multi-level domains in a couple of places (see the definition of multi-level domains in Subsection 42). The first place is in domain `beam`, in which attribute `critical_cross_section` is of type `cross_section`, a multi-level domain. Multi-level domain cross_section has two levels: `detailed` (of type `detailed_I_section`), and `simple` (of type `simple_I_section`). These two levels represent the levels of detail in which the critical cross-section of the beam can be idealized. The detailed I section (Figure 83-6), representing the actual design feature, takes into account the variation in thickness of the flange, whereas the simple I section (Figure 83-7) assumes a straight flange. This illustrates that the APM can simultaneously represent one or more idealized views of this design feature.

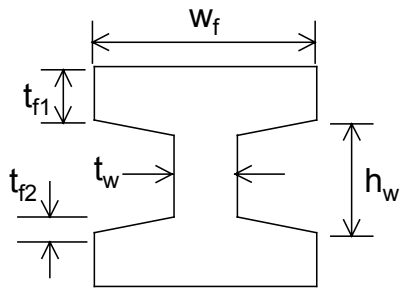


Figure 83-6: Critical Cross Section (Detailed)

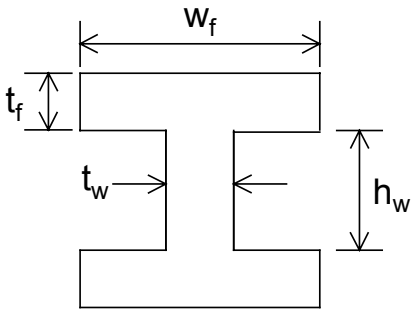


Figure 83-7: Critical Cross Section (Simple)

Relations pir5 through pir9, defined as follows:

```
pir5 : "<critical_cross_section.detailed.tf> == <tf>";
pir6 : "<critical_cross_section.detailed.tw> == <tw>";
pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
pir8 : "<critical_cross_section.detailed.wf> == <wf>";
pir9 : "<critical_cross_section.detailed.hw> == <hw>";
```

relate product attributes tf, tw, t2f, wf and hw of the shaft (design attributes) with attributes tf, tw, t2f, wf and hw (idealized attributes) of one of the levels (detailed) of critical_cross_section. Next, relations pir10 through pir13 define how the attributes of level simple of the multi-level domain cross_section are related to the attributes of level detailed:

```
pir10 : "<detailed.wf> == <simple.wf>";
pir11 : "<detailed.hw> == <simple.hw>";
pir12 : "<detailed.tf> == <simple.tf>";
pir13 : "<detailed.tw> == <simple.tw>";
```

As is the case in this multi-level domain, the types of the different levels of a multi-level domain may be subtyped from the same domain (although they do not *have* to). Here, the type of level detailed is detailed_I_section (a subtype of I_section) and the type of level simple is simple_I_section (also a subtype of I_section). The type of one level could even be a subtype of the type of another level. This allows sharing attributes among multiple levels of a multi-level domain. However, the relations in which a shared attribute participates may vary depending on the level. For example, the value of the area of the critical cross section of the beam is calculated differently depending on whether a detailed or a simple cross section is selected. When the simple level is selected, relation pir14:

```
pir14: "<area> == 2*<wf>*<tf> + <tw>*<hw>";
```

is used to calculate the area, whereas when the detailed level is selected, relation pir15:

```
pir15: "<area> == <wf>*( <t1f> + <t2f> ) + <tw>*( <t2f> - <t1f> ) +
        <tw>*<hw>";
```

is used instead.

The other place where multi-level domains are used in this APM is in domain material (in the second source set – flap_link_material_properties). Here, attribute stress_-strain_model is of type material_levels, a multi-level domain with two levels;

temperature_independent_linear_elastic and temperature_dependent_linear_-elastic.

The source set link defined in this APM is:

```
flap_link_geometric_model.flap_link.material ==
flap_link_material_properties.material.name;
```

This source set link joins instances of flap_link from the first source set with instances of material from the second source set when the value of attribute material of flap_link (a STRING) is equal to the value of attribute name of the material (also a STRING). When a match is found, the string to which attribute material of flap_link was pointing is replaced with the matching instance of material from the second source set.

The corresponding constraint schematics, EXPRESS-G and constraint network diagrams of this APM are shown in Figures 83-8, 83-9, 83-10 and 83-11.

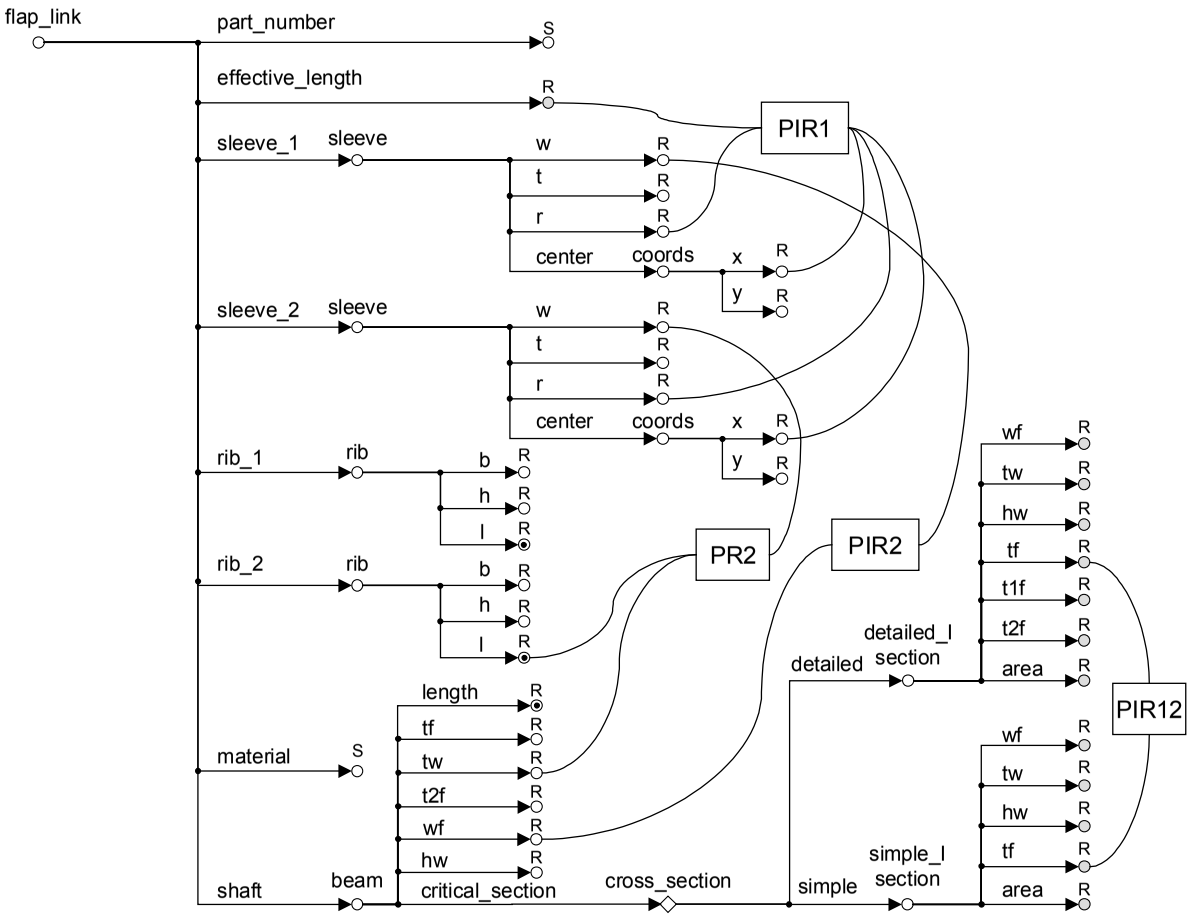


Figure 83-8: Flap Link APM Constraint Schematics Diagram (not all relations shown)

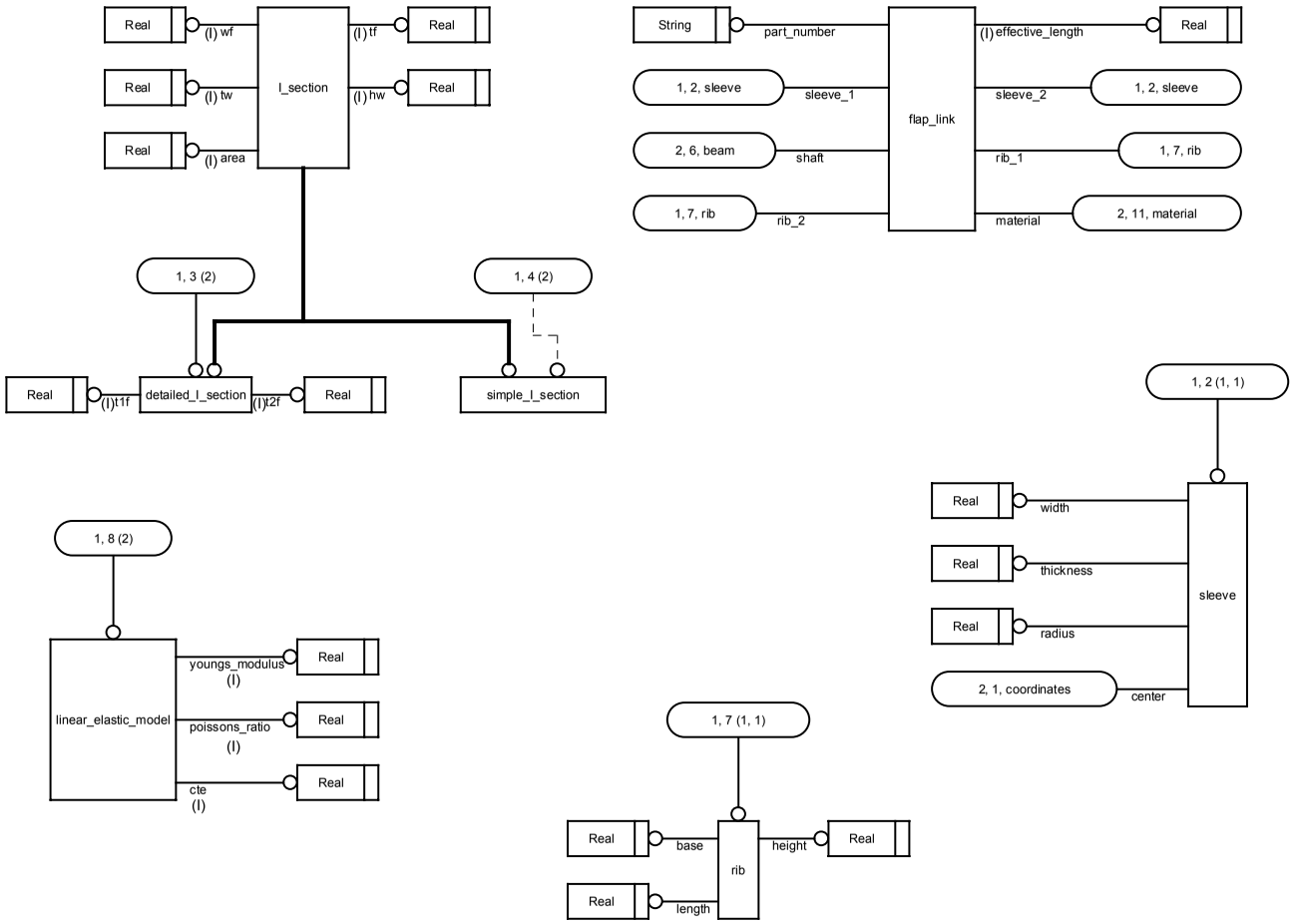


Figure 83-9: Flap Link APM EXPRESS-G Diagram

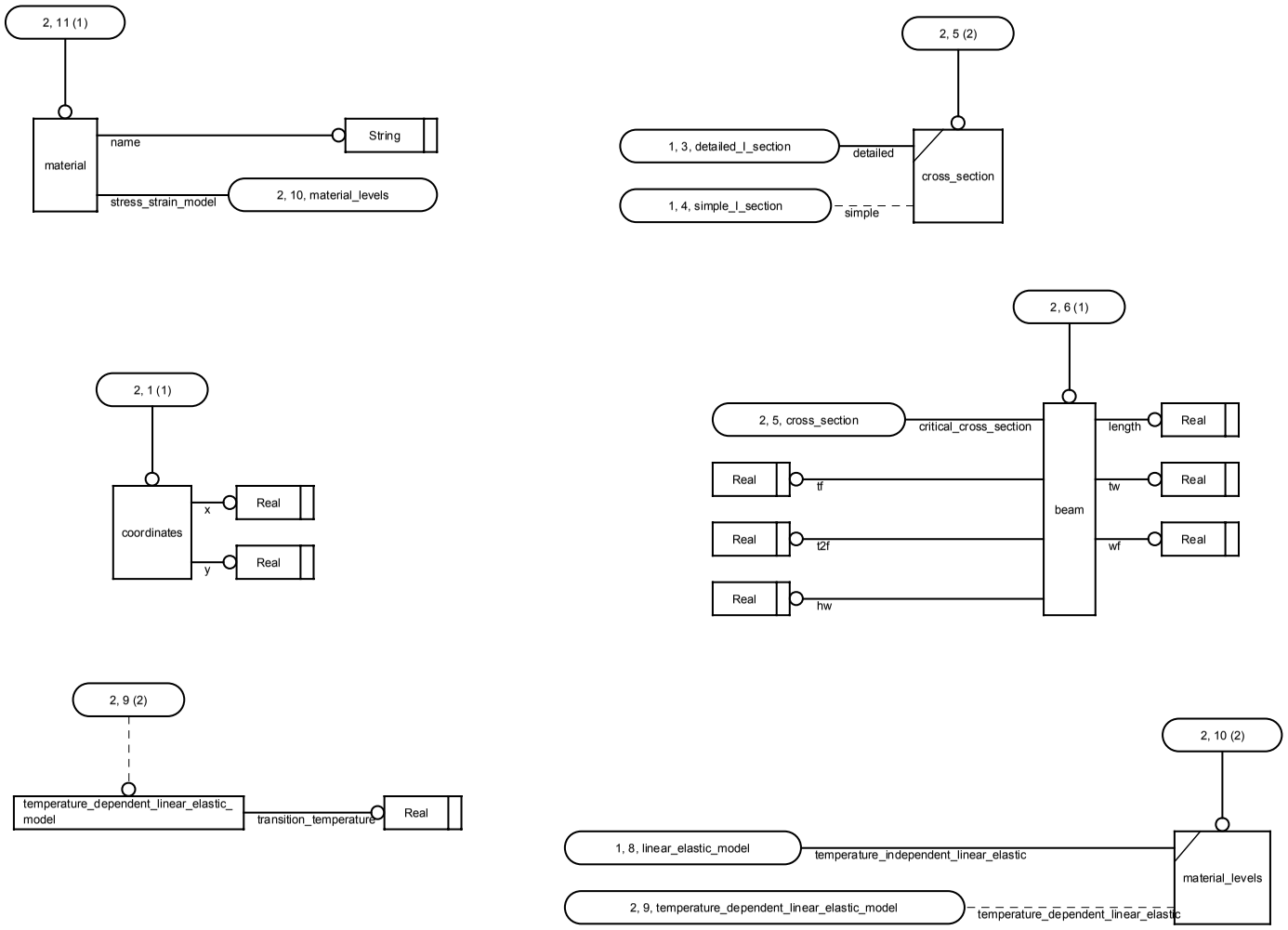


Figure 83-10: Flap Link APM EXPRESS-G Diagram (continued)

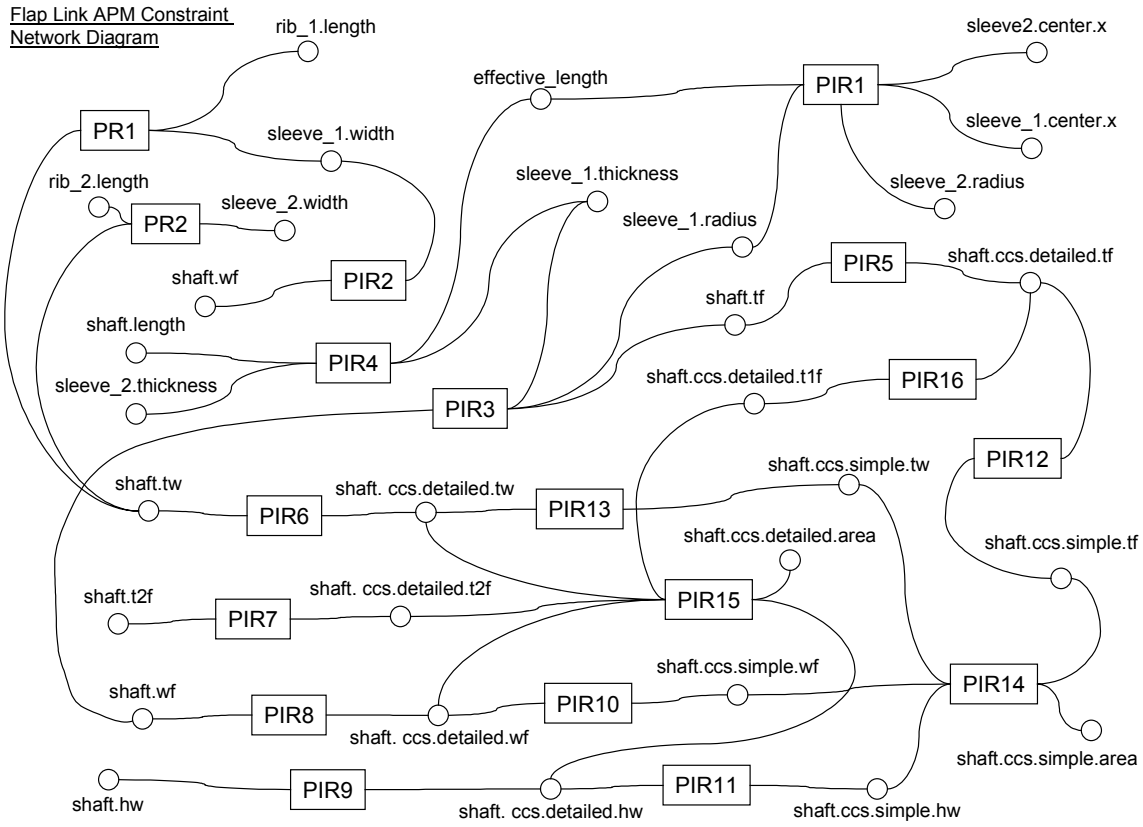


Figure 83-11: Flap Link APM Constraint Network Diagram

For each source set in an APM, there will be a repository or data file that contains instances of the domains defined in the source set. More specifically, these repositories contain instances of the *root* domain (or any of its subtypes) of each source set. In this example, there are two repositories: one corresponding to source set `flap_link_geometric_model` and the other to source set `flap_link_material_properties`. The first repository contains instances of `flap_link` (the root domain of source set `flap_link_-geometric_model`), and the second instances of `material` (the root domain of source set `flap_link_material_properties`).

Figure 83-12 shows an example of an APM-I file which defines two instances of domain `flap_link`: the first instance with `part_number` equal to `"FLAP-001"` and the second with `part_number` equal to `"FLAP-002"`. Figure 83-13 shows a second APM-I file which

defines three instances of domain `material`: the first with `name` equal to "`steel`", the second with `name` equal to "`aluminum`", and the third with `name` equal to "`cast iron`". Recall from Subsection 53 that instances of a domain are defined in APM-I by listing the attributes of the domain along with their values. If a value is unknown, a question mark ("`?`") is used instead of the value. For example, attribute `sleeve_2.center.x` in the first instance of `flap_link` (Figure 83-12) does not have value and therefore has a "?" instead of a value. A constraint-solving attempt using the appropriate relations will be initiated at run time when this value is queried by an APM client application. In general, the value of any type of attribute (idealized, essential or redundant – see Subsection 47) may be unknown. For example, the x coordinate of the center of sleeve 2 (`sleeve_2.center.x`) in the first instance of `flap_link` is unknown (despite being defined as `ESSENTIAL` in the APM). Notice also that attribute values that are known in one instance may be unknown in another instance of the same domain. For example, the value of `sleeve_2.center.x` is unknown in the first instance of `flap_link` ("`FLAP-001`") but known in the second one ("`FLAP-002`").

```
DATA;

INSTANCE_OF flap_link;
  part_number : "FLAP-001";
  effective_length : 12.5;
  sleeve_1.width : 1.5;
  sleeve_1.thickness : 0.5;
  sleeve_1.radius : 0.5;
  sleeve_1.center.x : 0.0;
  sleeve_1.center.y : 0.0;
  sleeve_2.width : 2.0;
  sleeve_2.thickness : 0.6;
  sleeve_2.radius : 0.75;
  sleeve_2.center.x : ?;
  sleeve_2.center.y : 0.0;
  shaft.length : ?;
  shaft.tf : 0.1;
  shaft.tw : 0.1;
  shaft.t2f : 0.15;
  shaft.wf : ?;
  shaft.hw : ?;
  shaft.critical_cross_section.detailed.wf : ?;
  shaft.critical_cross_section.detailed.tf : ?;
  shaft.critical_cross_section.detailed.tw : ?;
  shaft.critical_cross_section.detailed.hw : ?;
  shaft.critical_cross_section.detailed.area : ?;
  shaft.critical_cross_section.detailed.t1f : ?;
  shaft.critical_cross_section.detailed.t2f : ?;
  shaft.critical_cross_section.simple.wf : ?;
  shaft.critical_cross_section.simple.tf : ?;
  shaft.critical_cross_section.simple.tw : ?;
  shaft.critical_cross_section.simple.hw : ?;
  shaft.critical_cross_section.simple.area : ?;
  rib_1.base : 10.00;
  rib_1.height : 0.5;
  rib_1.length : ?;
  rib_2.base : 10.00;
  rib_2.height : 0.5;
  rib_2.length : ?;
  material : "aluminum";
END_INSTANCE;

INSTANCE_OF flap_link;
  part_number : "FLAP-002";
  effective_length : ?;
  sleeve_1.width : 1.5;
  sleeve_1.thickness : 0.5;
  sleeve_1.radius : 0.5;
  sleeve_1.center.x : 0.0;
  sleeve_1.center.y : 0.0;
  sleeve_2.width : 2.0;
  sleeve_2.thickness : 0.6;
  sleeve_2.radius : 0.75;
  sleeve_2.center.x : 20.00;
  sleeve_2.center.y : 0.0;
  shaft.length : ?;
  shaft.tf : 0.1;
  shaft.tw : 0.1;
  shaft.t2f : 0.15;
  shaft.wf : ?;
  shaft.hw : ?;
  shaft.critical_cross_section.detailed.wf : ?;
  shaft.critical_cross_section.detailed.tf : ?;
  shaft.critical_cross_section.detailed.tw : ?;
  shaft.critical_cross_section.detailed.hw : ?;
  shaft.critical_cross_section.detailed.area : ?;
  shaft.critical_cross_section.detailed.t1f : ?;
  shaft.critical_cross_section.detailed.t2f : ?;
  shaft.critical_cross_section.simple.wf : ?;
  shaft.critical_cross_section.simple.tf : ?;
  shaft.critical_cross_section.simple.tw : ?;
  shaft.critical_cross_section.simple.hw : ?;
  shaft.critical_cross_section.simple.area : ?;
  rib_1.base : 10.00;
  rib_1.height : 0.5;
  rib_1.length : ?;
  rib_2.base : 10.00;
  rib_2.height : 0.5;
  rib_2.length : ?;
  material : "steel";
END_INSTANCE;

END_DATA;
```

Figure 83-12: Flap Link Instances (APM-I Format)

```
DATA;

INSTANCE_OF material;
  name : "steel";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 30000000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.30;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.0000065;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 275.00;
END_INSTANCE;

INSTANCE_OF material;
  name : "aluminum";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 10400000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.000013;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 156.00;
END_INSTANCE;

INSTANCE_OF material;
  name : "cast iron";
  stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 18000000.00;
  stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
  stress_strain_model.temperature_independent_linear_elastic.cte : 0.000006;
  stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 125.00;
END_INSTANCE;


END_DATA;
```

Figure 83-13: Material Instances (APM-I Format)

Figure 83-14 shows the STEP P21 version of the two instances of `flap_link` defined in the APM-I file of Figure 83-12[50]. Correspondingly, Figure 83-15 shows the STEP P21 version of the three instances of `material` defined in the APM-I file of Figure 83-13.

[50] In this work, a value of –999.00 in a STEP P21 file represents an "unknown" value, even though the STEP standard (ISO 10303-21 1994) specifies that the symbol for unknown values is a question mark ("?"). However, the STEP development toolkit used for this work interprets a value of "?" as zero (which is more likely to be a *known* value versus –999.00).

```
DATA;
#10=FLAP_LINK('FLAP-001',12.5,#20,#40,#60,#100,#110,'aluminum');
#20=SLEEVE(1.5,0.5,0.5,#30);
#30=COORDINATES(0.0,0.0);
#40=SLEEVE(2.0,0.6,0.75,#50);
#50=COORDINATES(-999.00,0.0);
#60=BEAM(#70,-999.00,0.1,0.1,0.15,-999.00,-999.00);
#70=CROSS_SECTION(#80,#90);
#80=DETAILED_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00,-999.00,-999.00);
#90=SIMPLE_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00);
#100=RIB(10.0,0.5,-999.00);
#110=RIB(10.0,0.5,-999.00);

#120=FLAP_LINK('FLAP-002',-999.00,#130,#150,#170,#210,#220,'steel');
#130=SLEEVE(1.5,0.5,0.5,#140);
#140=COORDINATES(0.0,0.0);
#150=SLEEVE(2.0,0.6,0.75,#160);
#160=COORDINATES(20.00,0.0);
#170=BEAM(#180,-999.00,0.1,0.1,0.15,-999.00,-999.00);
#180=CROSS_SECTION(#190,#200);
#190=DETAILED_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00,-999.00,-999.00);
#200=SIMPLE_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00);
#210=RIB(10.0,0.5,-999.00);
#220=RIB(10.0,0.5,-999.00);

ENDSEC;
```

Figure 83-14: Flap Link Instances (STEP P21 Format)

```
DATA;
#10=MATERIAL('steel',#11);
#11=MATERIAL_LEVELS(#12,#13);
#12=LINEAR_ELASTIC_MODEL(30000000.00,0.30,0.0000065);
#13=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(275.00);

#110=MATERIAL('aluminum',#111);
#111=MATERIAL_LEVELS(#112,#113);
#112=LINEAR_ELASTIC_MODEL(10400000.00,0.25,0.000013);
#113=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(156.00);

#210=MATERIAL('cast iron',#211);
#211=MATERIAL_LEVELS(#212,#213);
#212=LINEAR_ELASTIC_MODEL(18000000.00,0.25,0.000006);
#213=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(125.00);

ENDSEC;
```

Figure 83-15: Material Instances (STEP P21 Format)

These data files (either in APM-I or in STEP P21 format) may be used by one or more APM client applications to perform specific tasks. For example, they can be used by the Flap Link Extension Analysis Application (presented in Subsection 91) to calculate the elongation and stress caused in the flap link when it is subjected to a tensional force. Alternatively, the two data files could be loaded into the APM Browser (presented in Subsection 93) to display the values of the various attributes of the flap link. Figures 83-16 and 83-17 show the output of the APM Browser displaying the attribute values of the `flap_link` instances. Notice that the APM Browser also outputs values that were not originally populated in the data files. These values were calculated at run time from the values of other attributes, using the relations defined in the APM. For example, recall that `sleeve_2.center.x` of flap link "`FLAP-001`" did not have a value in the data file of Figure 83-12, but the browser displays a value for it of 13.75 in. This value was calculated by the constraint solver using relation `pir1` (see the APM definition in Figure 83-5) and the values of `effective_length` (12.5 in), `sleeve_1.center.x` (0.0 in), `sleeve_1.radius` (0.5 in) and `sleeve_2.radius` (0.75 in) as follows:

```
sleeve_2.center.x = effective_length + sleeve_1.center.x +
   sleeve_1.radius + sleeve_2.radius

= 12.5 + 0.0 + 0.5 + 0.75 = 13.75 in
```

Notice that the fact that `effective_length` is in the left-hand side of relation `pir1` (Figure 83-5) does not imply that `effective_length` must always be the output (in this example `sleeve_2.center.x` is the output).

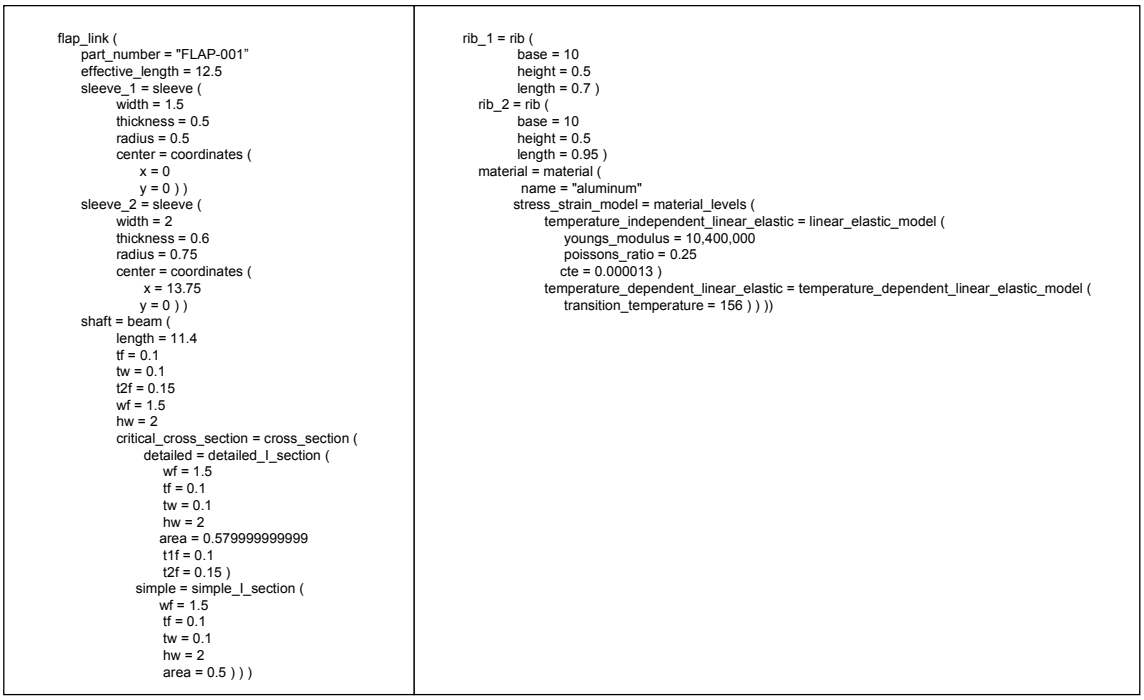


Figure 83-16: APM Browser Output (Flap Link instance "FLAP-001")

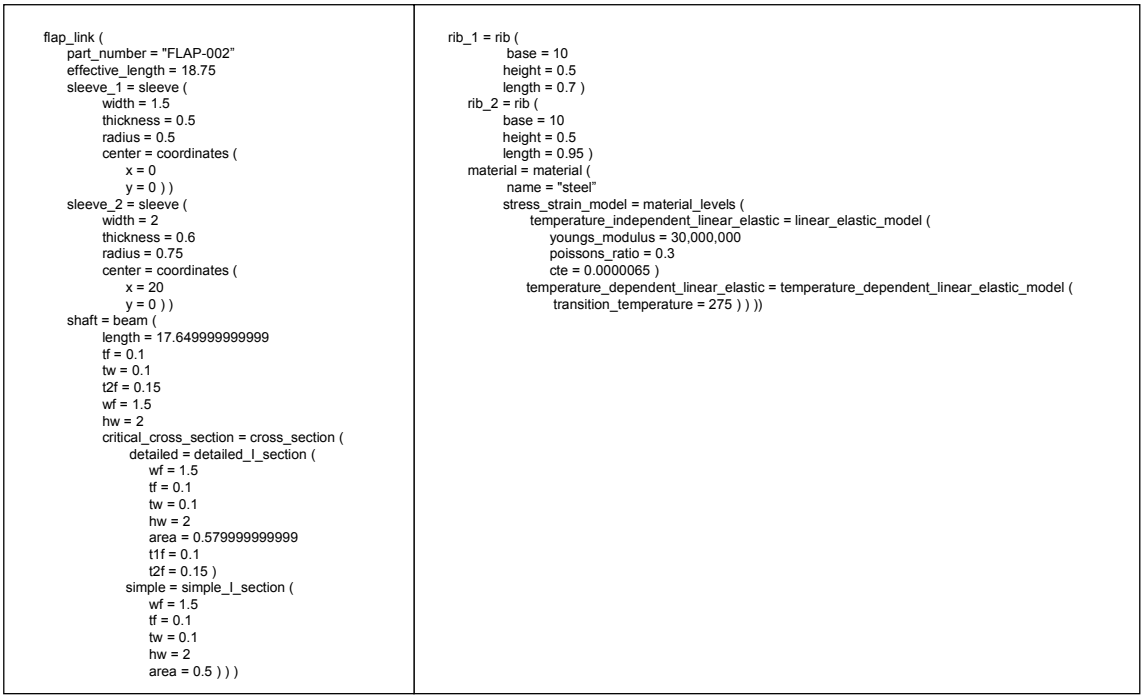


Figure 83-17: APM Browser Output (Flap Link) (Flap Link instance "FLAP-002")

Back Plate APM

The Back Plate APM describes a simple fictitious part consisting of a plate with two holes (Figure 83-18). The holes may have different diameters but must be aligned and centered with respect to the width of the plate. In addition, the diameter of the first hole (hole 1) must be greater than the diameter of the second hole (hole 2).

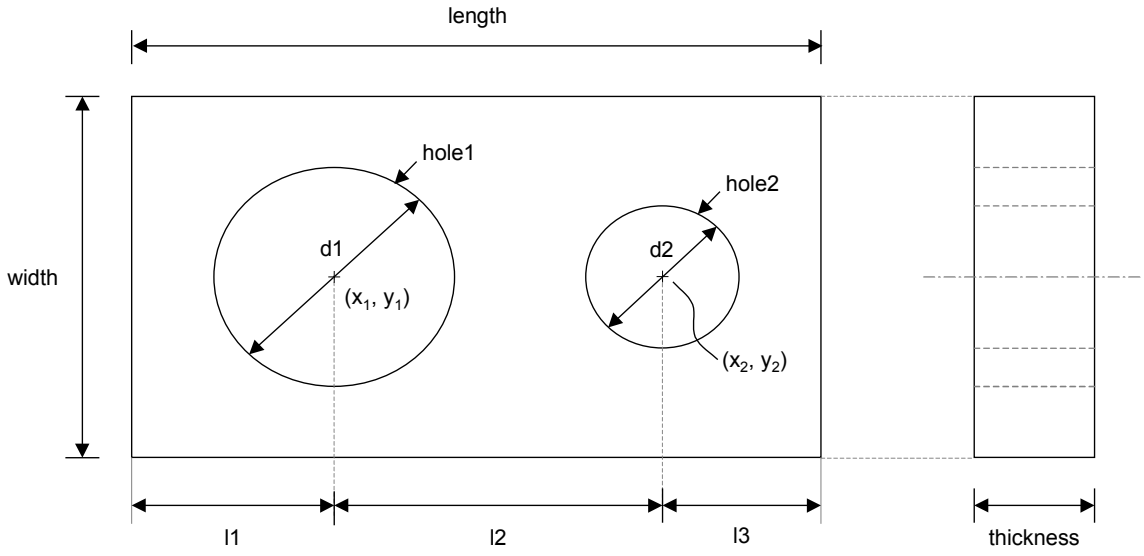


Figure 83-18: Back Plate

The APM-S definition for the back plate is shown in Figure 83-19. The corresponding constraint schematics and constraint network diagrams are shown in Figures 83-20 and 83-21, respectively. This APM contains three source sets: source set `back_plate_-geometric_model`, which contains entities to define the geometry of the plate; source set `back_plate_material_data`, which contains entities to define the material of which the plate is made; and source set `back_plate_employee_data` which contains entities to define information about people (thus simulating a very simple database of employees in an organization).

```
APM my_apm;


(* Back Plate Test Case APM *)

SOURCE_SET back_plate_geometric_model ROOT_DOMAIN plate;



DOMAIN part;
  part_number : STRING;
  designer : STRING;
END_DOMAIN;

DOMAIN plate SUBTYPE_OF part;
  l1 : REAL;
  l2 : REAL;
  l3 : REAL;
  ESSENTIAL width : REAL;
  ESSENTIAL length : REAL;
  ESSENTIAL thickness : REAL;
  hole1 : hole;
  hole2 : hole;
  ESSENTIAL material : STRING;
  IDEALIZED critical_area : REAL;

 PRODUCT_IDEALIZATION_RELATIONS
  pir_1 : "<critical_area> == ( <width> - <hole1.diameter> ) * <thickness>";
  pir_2 : "<hole1.center.y> == <width>/2";
  pir_3 : "<hole2.center.y> == <width>/2";
  pir_4 : "<l1> == <hole1.center.x>";
  pir_5 : "<l2> == <hole2.center.x> - <l1>";

 PRODUCT_RELATIONS
  pr_1 : "<length> == <l1> + <l2> + <l3>";

END_DOMAIN;

DOMAIN hole;
  ESSENTIAL diameter : REAL;
  area : REAL;
  center : coordinate;
 PRODUCT_RELATIONS
  pr_2: "<area> == Pi * <diameter>^2 / 4";
END_DOMAIN;
```

```
DOMAIN coordinate;
  ESSENTIAL x : REAL;
  ESSENTIAL y : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET back_plate_material_data ROOT_DOMAIN material;

DOMAIN material;
  materialName : STRING;
  ESSENTIAL youngsModulus : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET back_plate_employee_data ROOT_DOMAIN person;

DOMAIN person;
  first_name : STRING;
  last_name  : STRING;
  ssn : STRING;
END_DOMAIN;

END_SOURCE_SET;


LINK_DEFINITIONS
  back_plate_geometric_model.plate.material == back_plate_material_data.material.materialName;
  back_plate_geometric_model.part.designer == back_plate_employee_data.person.ssn;
END_LINK_DEFINITIONS;


END_APM;
```

Figure 83-19: Back Plate APM Definition

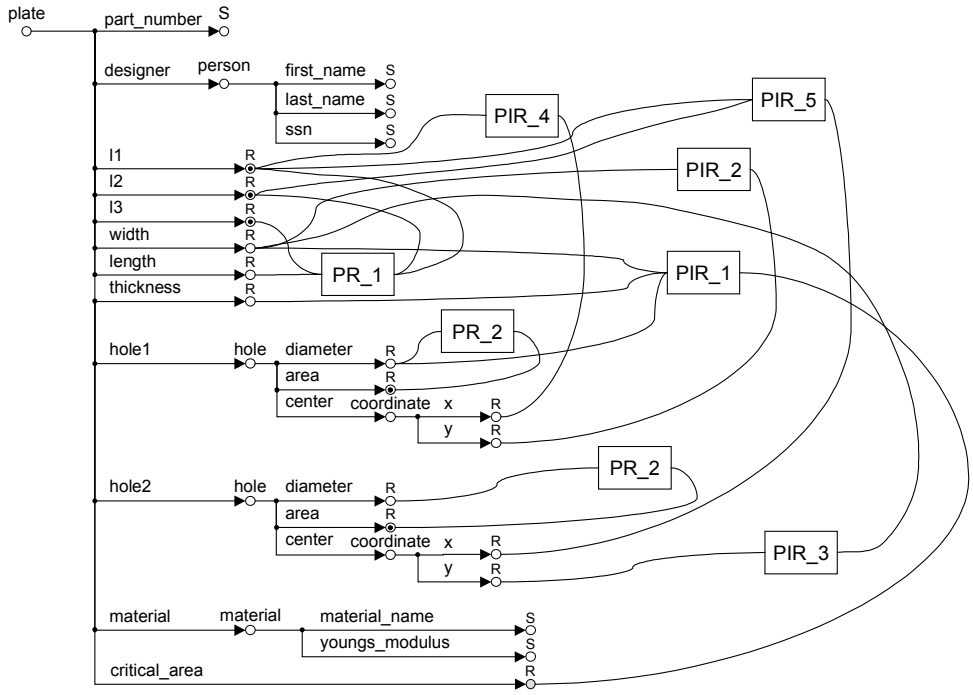


Figure 83-20: Back Plate Constraint Schematics Diagram

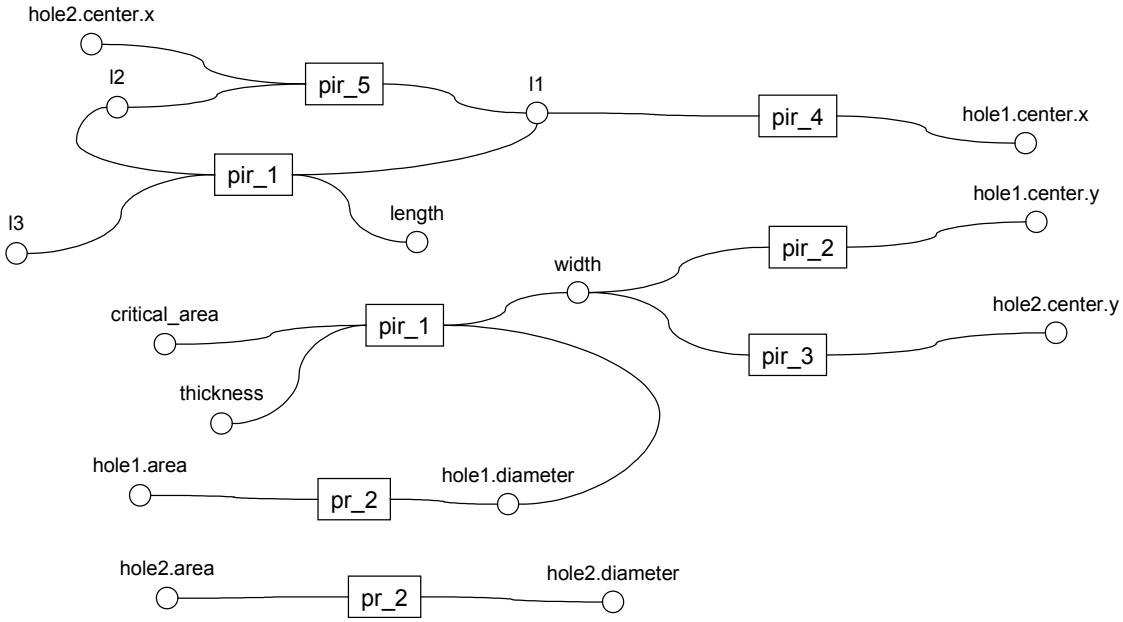


Figure 83-21: Back Plate Constraint Network Diagram

Domain `plate` is the root domain of the first source set. This domain defines the various dimensions of the plate (`l1`, `l2`, `l3`, `width`, `length`, `thickness`), the two holes (`hole1` and `hole2`), the material (`material`), and an idealized attribute called `critical_area`, which is considered to be the transversal cross section of the plate with the smallest area (and therefore "critical" for stress analysis purposes). Since hole 1 is assumed to be bigger than hole 2, the critical area will always be at hole 1. Product idealization relation `pir_1` defines the relationship between `critical_area`, the `width` and `thickness` of the plate, and the diameter of `hole1` as follows:

```
pir_1 : "<critical_area> == ( <width> - <hole1.diameter> ) *
         <thickness>";
```

Since this APM has three source sets, there must be at least two link definitions to join the instances of each source set. The first of the two source set link definitions in this APM is:

```
back_plate_geometric_model.plate.material ==
   back_plate_material_data.material.materialName;
```

which links instances of `plate` from the first source set (`back_plate_geometric_model`) with instances of `material` from the second source set (`back_plate_material_data`)

when the value of attribute plate.material is equal to the value of attribute material.materialName. The second source set link definition is:

```
back_plate_geometric_model.part.designer ==
   back_plate_employee_data.person.ssn;
```

which links instances of plate from the first source set with instances of person from the third source set (back_plate_employee_data) when the value of attribute part.designer (part is the supertype of plate) is equal to the value of attribute person.ssn.

Figure 83-22 shows an example of an APM-I file (corresponding to source set back_plate_geometric_model) which defines two instances of domain plate: the first instance with part_number equal to "XYZ-001" and the second with part_number equal to "XYZ-002". Figure 83-23 shows a second APM-I file (corresponding to source set back_plate_material_data) which defines two instances of domain material: the first instance with materialName equal to "steel" and the second with materialName equal to "aluminum". Figure 83-24 shows a third APM-I file (corresponding to source set back_plate_employee_data), which defines two instances of domain person: the first with first_name equal to "Diego" and the second with first_name equal to "Patricia". Figure 83-25 shows the STEP P21 version of the APM-I definition shown in Figure 83-22.

```
DATA;

INSTANCE_OF plate;
part_number : "XYZ-001";
designer : "1234";
l1 : ?;
l2 : ?;
l3 : 5.00;
width : 20.00;
length : ?;
thickness : 0.25;
hole1.diameter : ?;
hole1.center.x : 10.00;
hole1.center.y : ?;
hole1.area : ?;
hole2.diameter : 6.00;
hole2.center.x : 20.00;
hole2.center.y : ?;
hole2.area : ?;
material : "steel";
critical_area : ?;
END_INSTANCE;
```

```
INSTANCE_OF plate;
part_number : "XYZ-002";
designer : "567";
l1 : ?;
l2 : ?;
l3 : ?;
width : 25.00;
length : 35.00;
thickness : 0.30;
hole1.diameter : 9.00;
hole1.center.x : 12.00;
hole1.center.y : ?;
hole1.area : ?;
hole2.diameter : 6.00;
hole2.center.x : 20.00;
hole2.center.y : ?;
hole2.area : ?;
material : "aluminum";
critical_area : ?;
END_INSTANCE;

INSTANCE_OF part;
part_number : "XYZ-001";
designer : "567";
END_INSTANCE;

END_DATA;
```

Figure 83-22: Back Plate Instances (APM-I Format)

```
DATA;

INSTANCE_OF material;
  materialName : "steel";
  youngsModulus : 3000000.00;
END_INSTANCE;

INSTANCE_OF material;
  materialName : "aluminum";
  youngsModulus : 10400000.00;
END_INSTANCE;

END_DATA;
```

Figure 83-23: Material Instances (APM-I Format)

```
DATA;

INSTANCE_OF person;
  first_name : "Diego";
  last_name  : "Tamburini";
  ssn : "1234";
END_INSTANCE;

INSTANCE_OF person;
  first_name : "Patricia";
  last_name  : "Esparza";
  ssn : "567";
END_INSTANCE;

END_DATA;
```

Figure 83-24: Person Instances (APM-I Format)

```
DATA;
#10=PLATE('XYZ-001','1234',-999.00,-999.00,5.00,20.00,-999.00,0.25,#20,#40,'steel',-999.00);
#20=HOLE(-999.00,-999.00,#30);
#30=COORDINATE(10.0,-999.00);
#40=HOLE(6.00,-999.00,#50);
#50=COORDINATE(20.0,-999.00);

#100=PLATE('XYZ-002','567',-999.00,-999.00,-999.00,25.00,35.00,0.30,#200,#40,'aluminum',-999.00);
#200=HOLE(9.0,-999.00,#300);
#300=COORDINATE(12.0,-999.00);
#400=HOLE(6.00,-999.00,#500);
#500=COORDINATE(20.0,-999.00);


ENDSEC;
END-ISO-10303-21;
```

Figure 83-25: Back Plate Instances (STEP P21 Format)

Finally, Figure 83-26 shows the output of the APM Browser using the instances from the data files above. Notice how some values that were not originally populated in the design file of Figure 83-22 (that is, those with a "?" instead of a value) were calculated and displayed by the APM Browser while others were not. For example, the APM Browser displays a value for the `critical_area` of plate "`XYZ-002`" of 4.799 in^2. However, the `critical_area` of plate "`XYZ-001`" could not be calculated (and therefore is displayed with "`No Value`") because there were not enough input values to calculate it using the relations defined in the APM. More specifically, the critical area of plate "`XYZ-001`" could not be calculated with

relation `pir_1` (Figure 83-19) because the diameter of `hole1` had no value either. The second instance of `plate` (plate "`XYZ-002`"), on the other hand, did have all the values needed to calculate the `critical_area`, and therefore it was possible to solve for its value.

```
plate (
    part_number = "XYZ-001"
    designer = person (
        first_name = "Diego"
        last_name = "Tamburini"
        ssn = "1234" )
    l1 = 10
    l2 = 10
    l3 = 5
    width = 20
    length = 25
    thickness = 0.25
    hole1 = hole (
        diameter = No value
        center = coordinate (
            x = 10
            y = 10 )
        area = No value )
    hole2 = hole (
        diameter = 6
        center = coordinate (
            x = 20
            y = 10 )
        area = 28.274333882308 )
    material = material (
        materialName = "steel"
        youngsModulus = 3,000,000 )
    critical_area = No value)
```

```
plate (
    part_number = "XYZ-002"
    designer = person (
        first_name = "Patricia"
        last_name = "Esparza"
        ssn = "567" )
    l1 = 11.999999999999
    l2 = 8
    l3 = 15
    width = 25
    length = 35
    thickness = 0.3
    hole1 = hole (
        diameter = 9
        center = coordinate (
            x = 12
            y = 12.5 )
        area = 63.617251235193 )
    hole2 = hole (
        diameter = 6
        center = coordinate (
            x = 20
            y = 12.5 )
        area = 28.274333882308 )
    material = material (
        materialName = "aluminum"
        youngsModulus = 10400000.00 )
    critical_area = 4.799999999999)
```

Figure 83-26: APM Browser Output (Back Plate)

Wing Flap Support APM[51]

The wing flap support (shown in Figure 83-27) is an assembly consisting of two beams (known as inboard and outboard beams) held together by bolts. As shown in Figure 83-28, its forward end is attached to the wing structure of an airplane (at the "underwing fitting" point in the figure). The aft end interfaces with a pivot link, which attaches to the flap carrier

[51] This APM was developed as a representative aerospace application for the Boeing Product Simulation Integration for Structures (PSI) project discussed in Chapter 2 (Peak, Fulton et al. 1999; Prather and Amador 1997).

beam, to which the actual wing flap is attached. The central portion of the support also attaches to the structure of the wing at the rear spar fitting. The entire support remains stationary with respect to the wing and provides structural support to the several pieces of the mechanism that provides upward and downward motion to the flap of the wing.

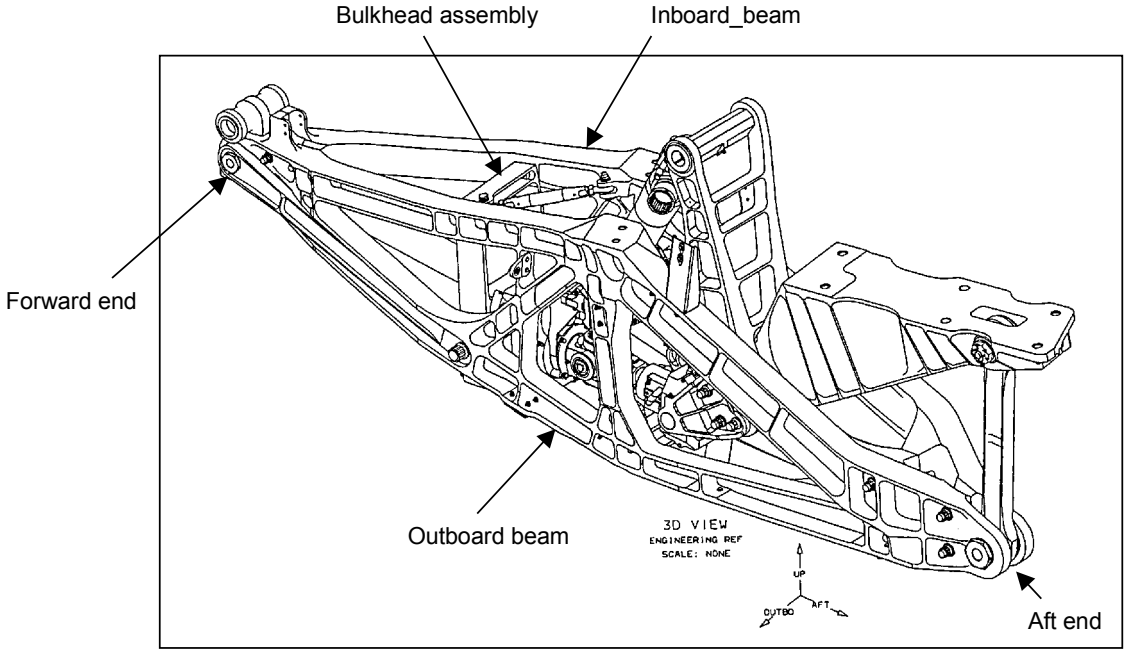


Figure 83-27: Wing Flap Support Assembly

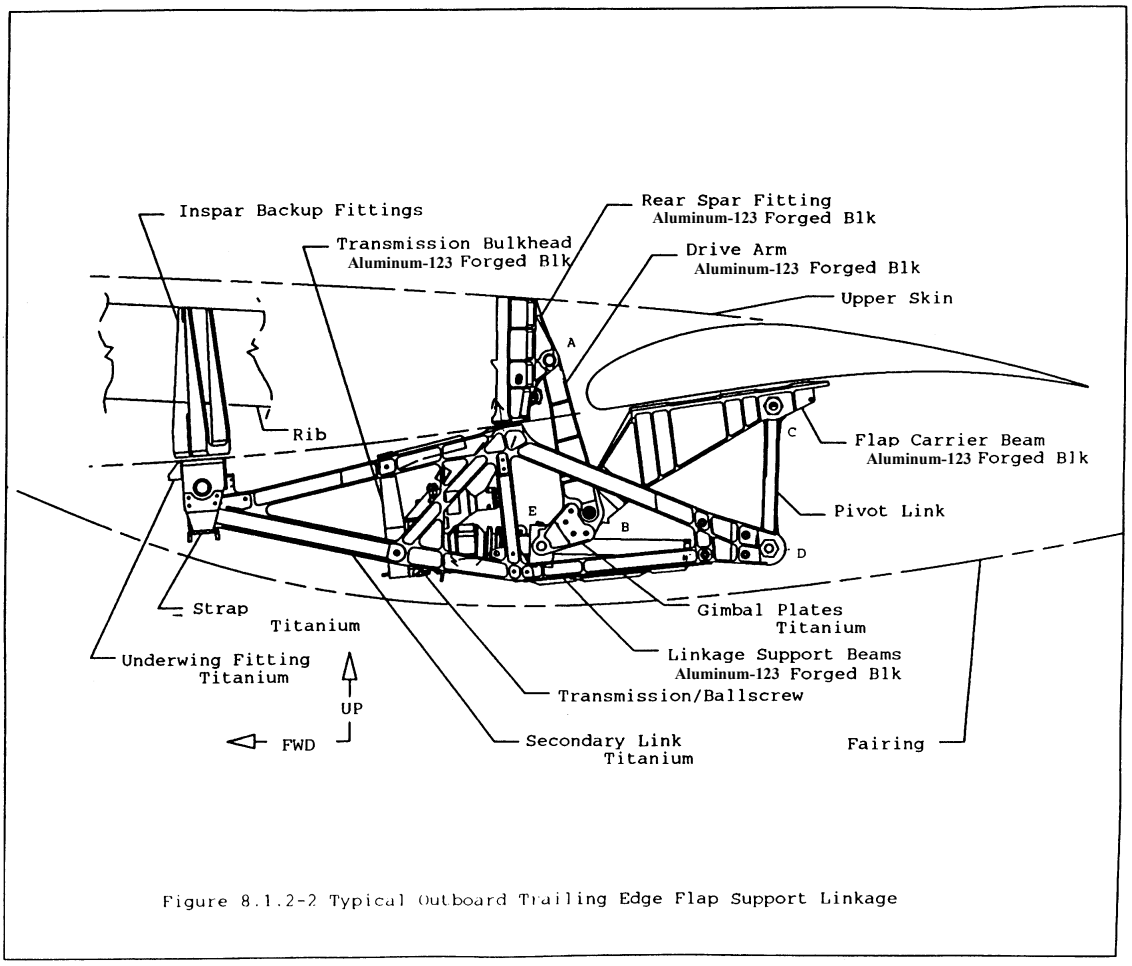


Figure 83-28: Wing Flap Mechanism

This APM focuses on the *inboard* beam of the wing flap support (also known as "bicycle frame" because of its shape) shown in Figures 83-29 and 83-30. As shown in Figure 83-29, the inboard beam is a single molded part whose features include seven legs (legs 1 to 7), three fittings (forward fitting, aft fitting and spar interface) and two joints (joint 5-6 - joining legs 5 and 6 - and joint 6-7 – joining legs 6 and 7). The legs have cavities through which holes are drilled in order to provide attachment points for the various fixtures that are attached to the beam (such as the bulkhead assembly, shown in Figure 83-27).

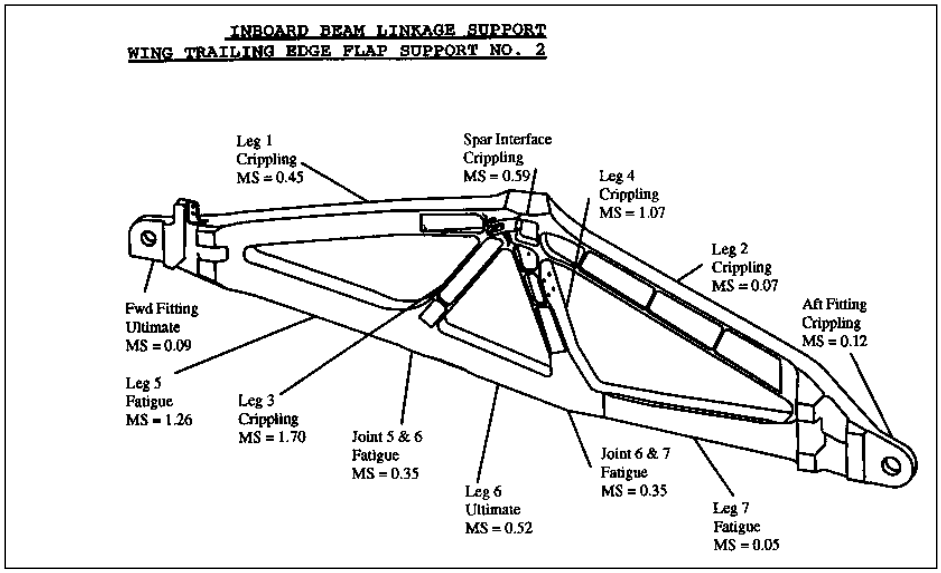


Figure 83-29: Inboard Beam of the Wing Flap Support Assembly

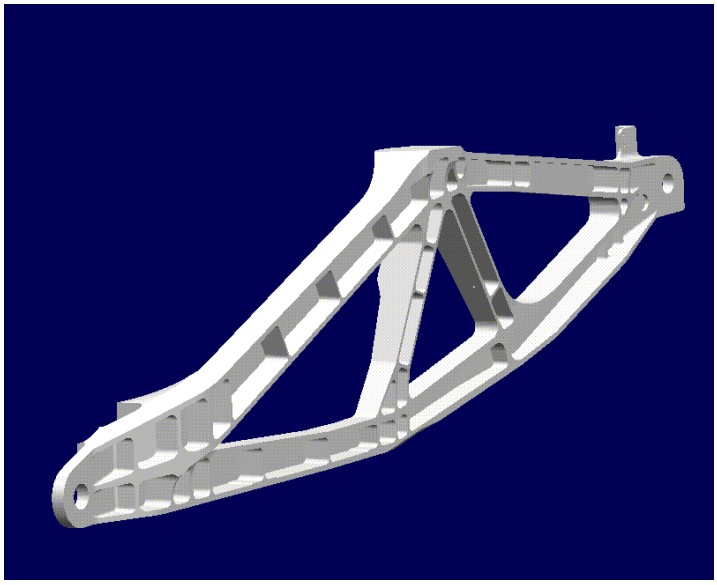


Figure 83-30: Inboard Beam of the Wing Flap Support Assembly (CAD 3D Model)

For the purpose of this discussion, only one leg ("Leg 1") of the inboard beam and only one of the cavities of this leg ("Cavity 3") will be considered (Figure 83-31). This section of the

leg is called "bulkhead attachment point" because one of the fasteners used to attach the bulkhead (shown in the lower portion of Figure 83-31) will be inserted through the hole drilled through the bottom surface of Cavity 3. The cavity is confined by two ribs, named "Rib 8" and "Rib 9". The dimensions of this cavity are shown in more detail in Figure 83-32.

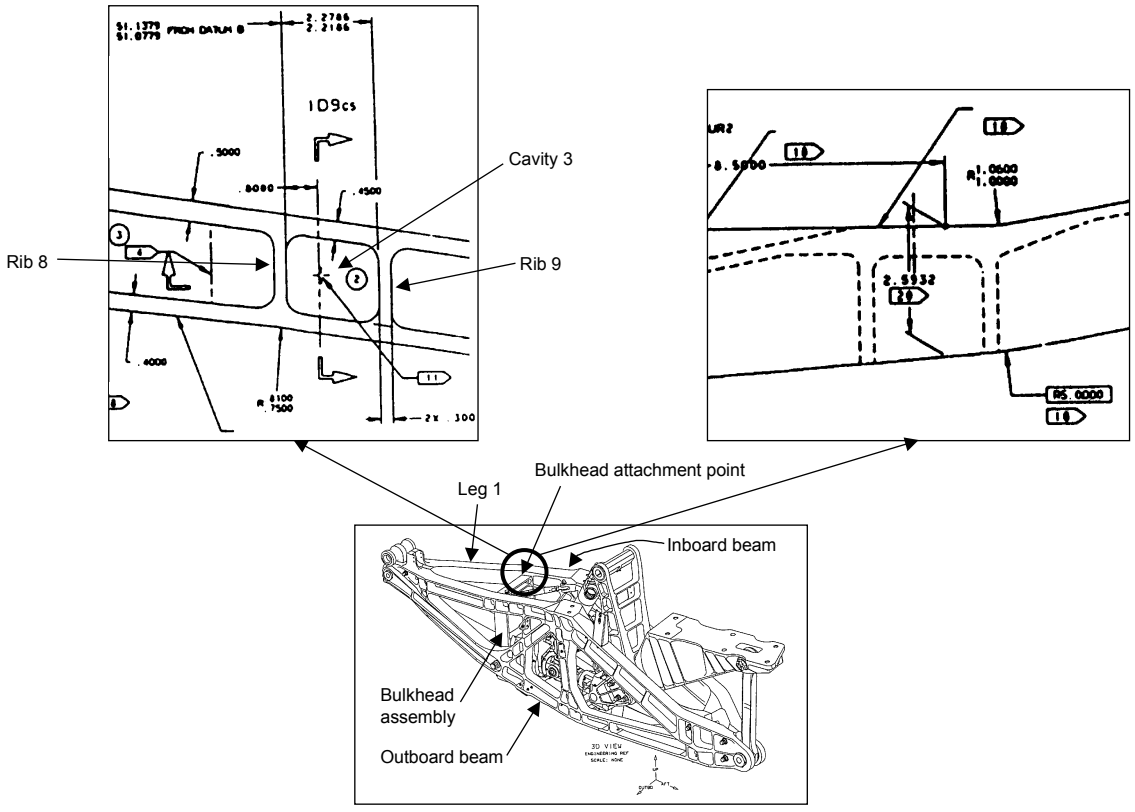


Figure 83-31: Bulkhead Attachment Point on Inboard Beam Leg 1

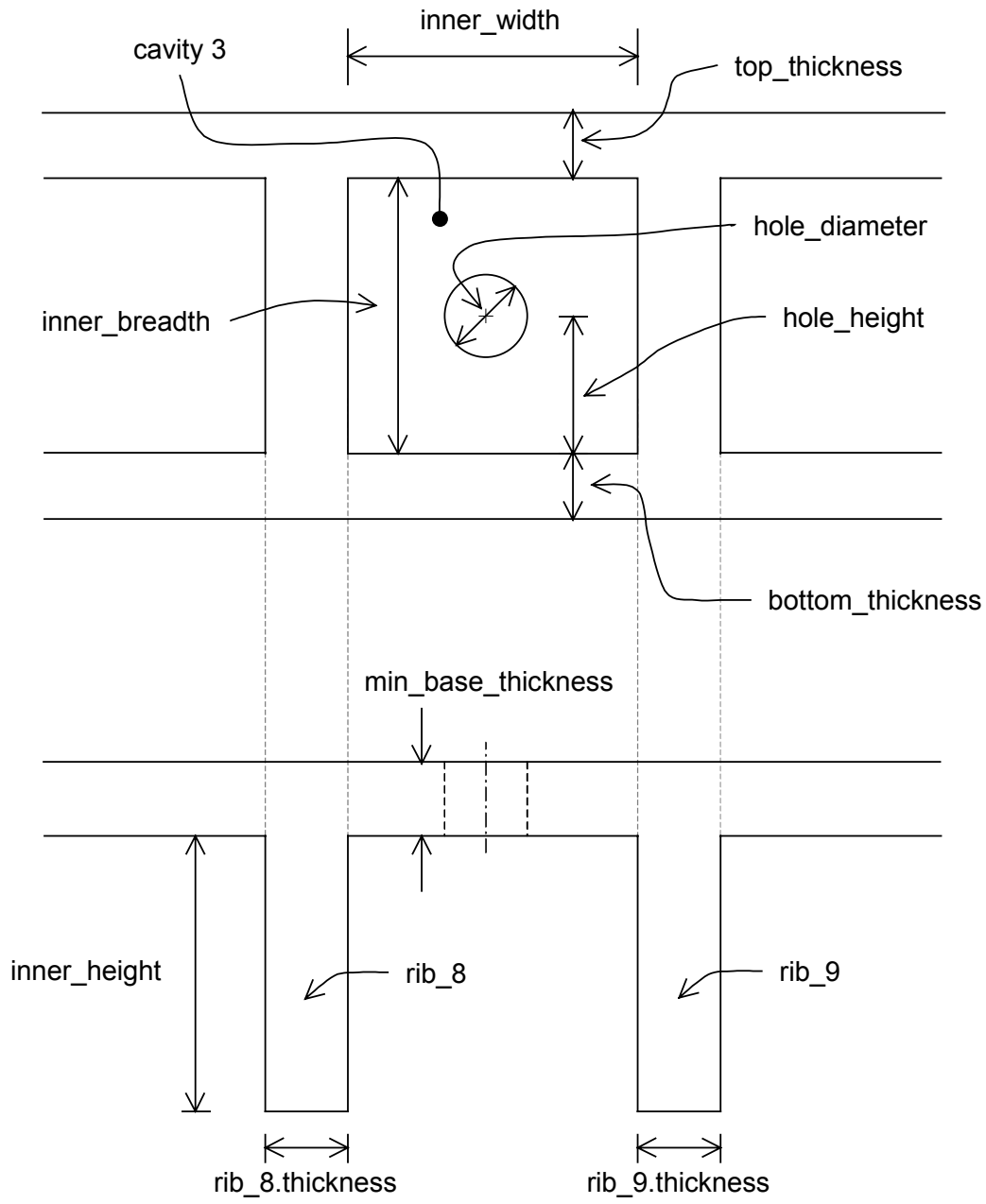


Figure 83-32: Dimensions of Cavity 3 of Leg 1

The APM corresponding to this partial inboard beam is shown in Figure 83-33 (the complete APM definition is provided in Appendix Z.1). Its corresponding constraint schematics and constraint network diagrams are shown in Figures 83-34 and 83-34, respectively.

```
APM simple_inboard_beam;

SOURCE_SET simple_inboard_beam ROOT_DOMAIN inboard_beam;

DOMAIN inboard_beam;
   leg_1 : leg;
END_DOMAIN;


DOMAIN leg;
   cavity_3 : cavity_with_bottom_hole;
   rib_8 : rib;
   rib_9 : rib;
   bulkhead_attach_point : channel_fitting;
 PRODUCT_IDEALIZATION_RELATIONS
   pir1: "<bulkhead_attach_point.end_pad.width> == <rib_8.thickness>/2 + <cavity_3.inner_width> + <rib_9.thickness>/2";
   pir2: "<bulkhead_attach_point.end_pad.height> == <cavity_3.bottom_thickness>/2 + <cavity_3.inner_breadth>";
   pir3: "<bulkhead_attach_point.end_pad.thickness> == <cavity_3.minimum_base_thickness>";
   pir4: "<bulkhead_attach_point.end_pad.hole_diameter> == <cavity_3.hole_diameter>";
   pir5: "<bulkhead_attach_point.end_pad.hole_center_height> == <cavity_3.hole_height> + <cavity_3.bottom_thickness>";
   pir6: "<bulkhead_attach_point.base.width> == <bulkhead_attach_point.end_pad.width>";
   pir7: "<bulkhead_attach_point.base.height> == <cavity_3.inner_height> + <cavity_3.minimum_base_thickness>/2";
   pir8: "<bulkhead_attach_point.base.thickness> == <cavity_3.bottom_thickness>";
   pir9: "<bulkhead_attach_point.base.hole_diameter> == 0";
   pir10: "<bulkhead_attach_point.base.hole_center_height> == 0";

   pir11: "<bulkhead_attach_point.wall.width> == <bulkhead_attach_point.base.height>";
   pir12: "<bulkhead_attach_point.wall.height> ==<bulkhead_attach_point.end_pad.height>";
   pir13: "<bulkhead_attach_point.wall.thickness> == ( <rib_8.thickness> + <rib_9.thickness> )/2";
END_DOMAIN;

DOMAIN cavity_with_bottom_hole;
   inner_width : REAL;
   inner_breadth : REAL;
   inner_height : REAL;
   minimum_base_thickness : REAL;
   top_thickness : REAL;
   bottom_thickness : REAL;
   hole_diameter : REAL;
   hole_height : REAL;
END_DOMAIN;

DOMAIN rib;
   thickness : REAL;
END_DOMAIN;

DOMAIN channel_fitting;
   end_pad : wall_with_hole;
   base : wall_with_hole;
   wall : wall;
END_DOMAIN;

DOMAIN wall;
   IDEALIZED width : REAL;
   IDEALIZED height : REAL;
   IDEALIZED thickness : REAL;
END_DOMAIN;

DOMAIN wall_with_hole SUBTYPE_OF wall;
   IDEALIZED hole_diameter : REAL;
   IDEALIZED hole_center_height : REAL;
END_DOMAIN;

END_SOURCE_SET;

END_APM;
```

Figure 83-33: Partial Inboard Beam APM

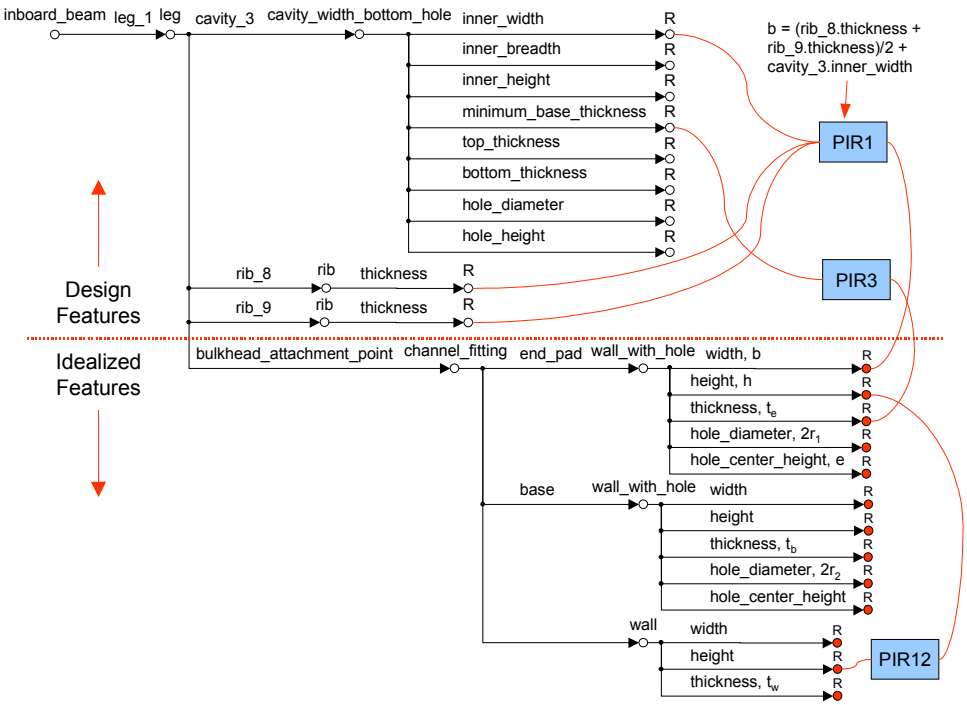


Figure 83-34: Partial Inboard Beam Constraint Schematics Diagram (not all relations shown)

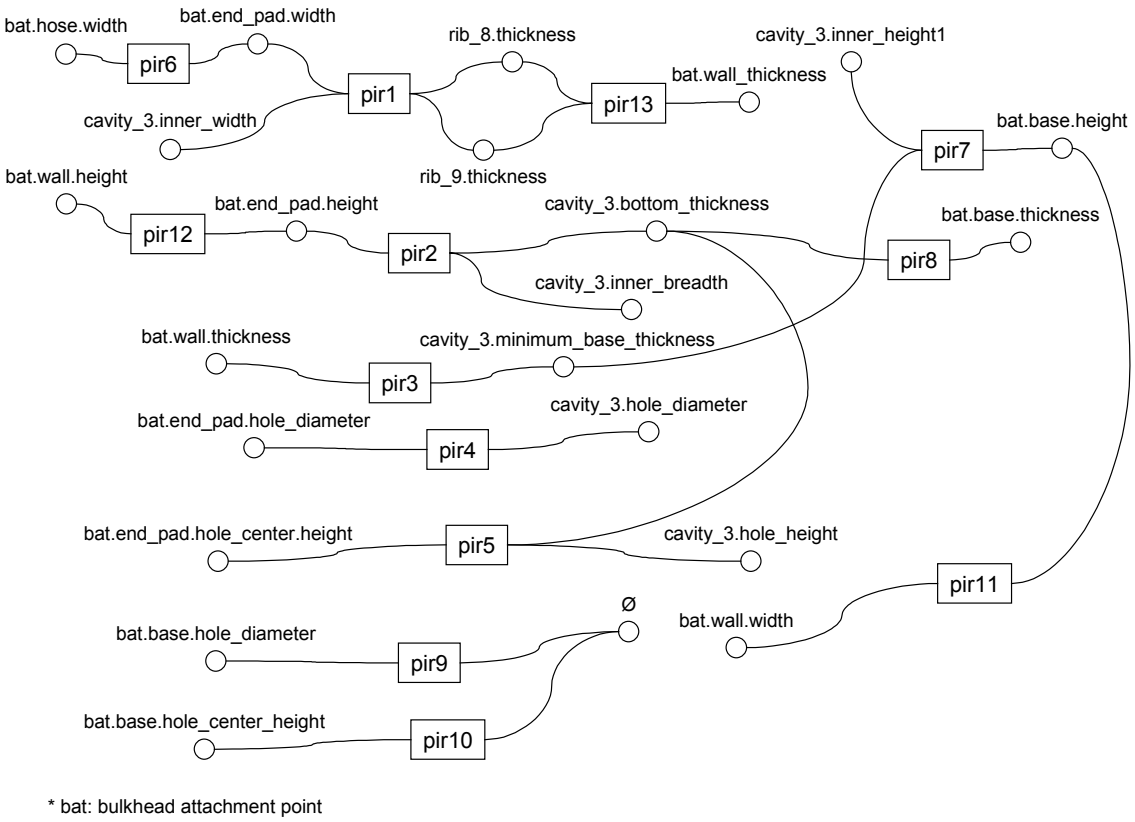


Figure 83-35: Partial Inboard Beam Constraint Network Diagram

Although this is a relatively simple APM, it demonstrates a key aspect of the APM representation: the ability to relate design features of a part to idealized features (used by analysis models) through product idealization relations. This ability is important because analysis models are often built in terms of an *idealized* version of a part or feature of a part. These models are often available in libraries of analysis models contained in design manuals and electronic templates compiled by companies, professional organizations or academic publications. They are normally well established, tested, and known to provide accurate results (as long as they are applied in the right situation and none of the assumptions and boundary conditions specified are being violated).

The test case for which this APM was developed provides an example of the need to relate design features to idealized features. In this scenario, an analyst wants to estimate the stresses and allowable loads in various critical points of the bulkhead attachment point of the inboard beam caused by the load transmitted by the fastener attaching the bulkhead. For this

purpose, he or she determines that it is appropriate to use an analysis template for generic channel fittings – such as the one shown in Figure 83-36 - already available (in electronic form) in his or her company. The analysis model on which this template is based idealizes the channel fitting as having a geometry such as the one shown in Figure 83-37.

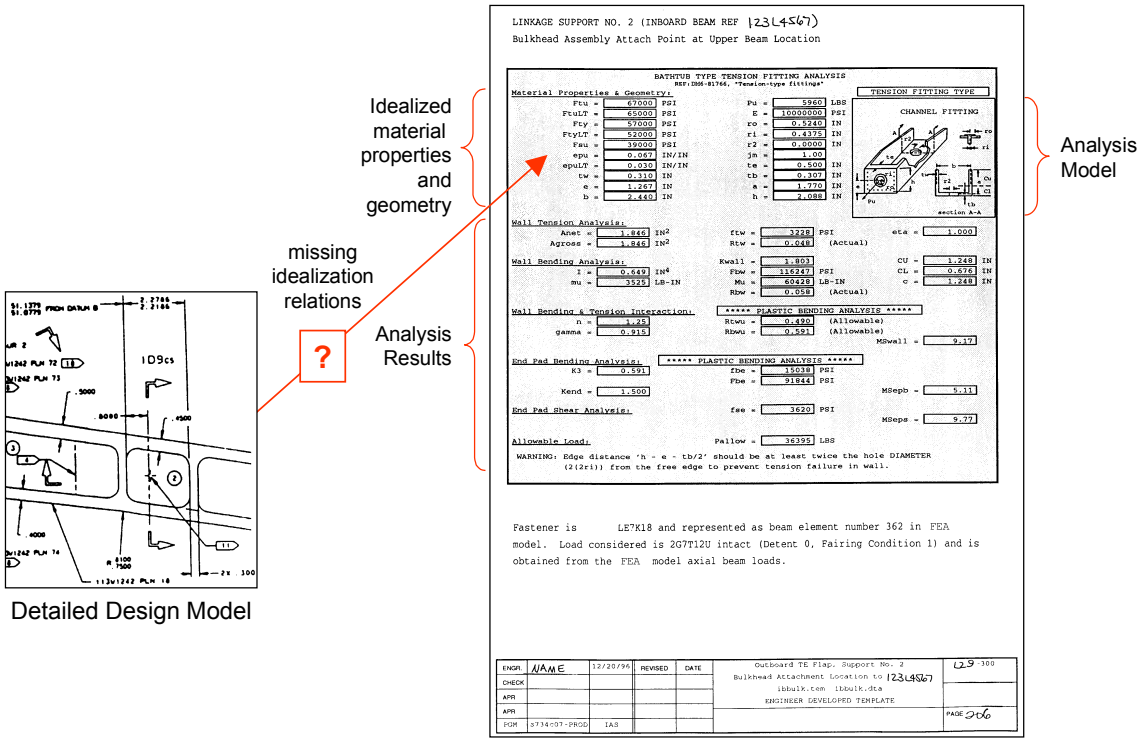


Figure 83-36: Channel Fitting Analysis Template

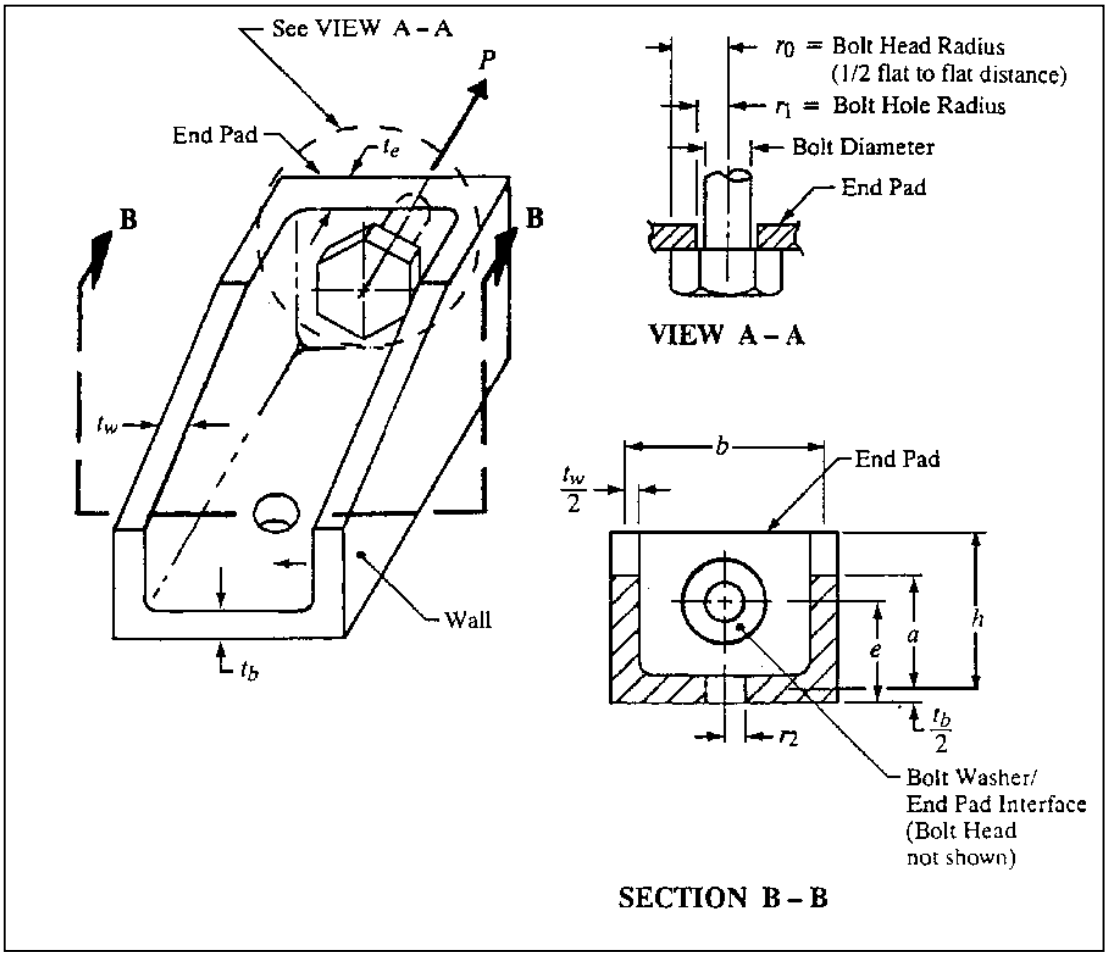


Figure 83-37: General Idealized Channel Fitting

Without the APM approach, the analyst would manually fill the input fields of this electronic template (in "Material Properties & Geometry" section of the template), execute the analysis and get the results displayed in the various output fields. The main disadvantage of this approach - in addition to the fact that the analyst would have to enter these values manually - is that he or she would have to manually retrieve the values from the design representation of the bulkhead attachment point, and transform and idealize them in order to put them in terms of the generic channel fitting model represented in this template. In most cases, this retrieval, transformation and idealization process is not captured or documented in any form.

To overcome this problem, the Inboard Beam APM of Figures 83-33 and 83-34 could be used as an intermediate representation between the design representation of the bulkhead attachment point and the generic channel fitting analysis representation supported by the template. This APM could be used to automatically extract, transform and idealize the design values and obtain the values required by the channel fitting analysis model. In this APM, the bulkhead attachment point has been idealized as a channel fitting. For this purpose, an idealized feature called `bulkhead_attach_point` of type `channel_fitting` was added as an attribute of the `leg`, and a set of product idealization relations (relations `pir1` through `pir13`) were defined to specify how the attributes of this idealized feature are obtained from the attributes of the "real" features of the part (the cavity and the two ribs of the bulkhead attachment point). For example, `pir1` (one of these product idealization relations):

```
pir1: "<bulkhead_attach_point.end_pad.width> == <rib_8.thickness>/2 +
       <cavity_3.inner_width> + <rib_9.thickness>/2";
```

specifies how the width of the end pad of the channel fitting (an idealized attribute) is related to the thicknesses of the two ribs and the inner width of the cavity (product attributes). This relation (as well as relation `pir3`) is shown graphically in Figure 83-38.

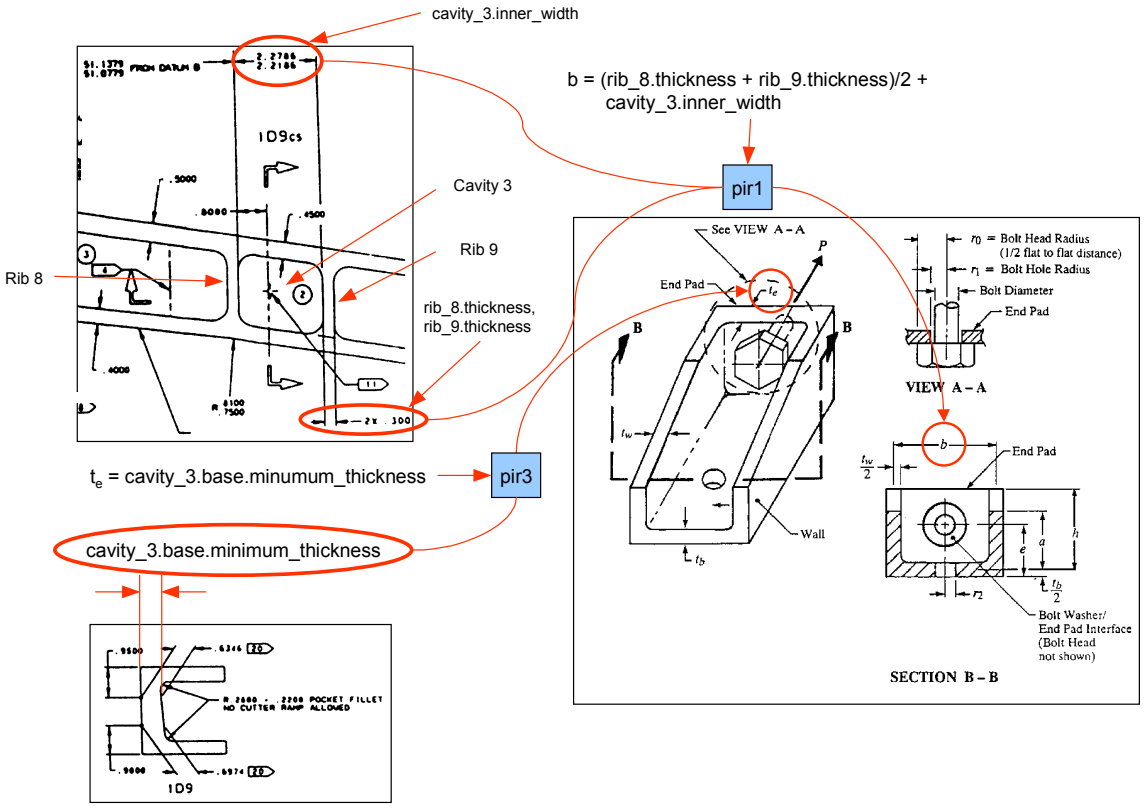


Figure 83-38: Relating Design to Idealized Features in the Channel Fitting Idealization of the Bulkhead Attachment Point

This type of information – how to connect design features to idealized features - is key to design-analysis integration, yet is rarely documented in the analysis documentation or explicitly captured anywhere. The analysis template of Figure 83-36 exemplifies this problem: the connections between the channel fitting variables contained in this template and those of the bulkhead attachment point are not explicitly represented. For instance, by just looking at this template it is very hard to determine that the value of variable **b** (2.440 in) of the channel fitting model is calculated with the thicknesses of ribs 8 and 9 and the inner width of cavity 3 of the bulkhead attachment point – as defined by relation `pir1` above. Hence, it difficult to reproduce the idealization decisions made by the analyst and automate the idealization process. One of the most significant contributions of the APM Representation is that it provides a mechanism to define these relations explicitly as product idealization relations.

In addition, domain definitions such as `channel_fitting` above can potentially be reused in multiple APMs. Only the relations connecting the attributes of these reusable domains to

the attributes of the part containing them would need to be customized for the part in question. Therefore, a library of commonly used APM domains (such as `channel_fitting`) could be maintained to facilitate reusability.

Figure 83-39 shows an instance of domain `inboard_beam` defined in APM-I. Notice that in this data file only *product attributes* (the attributes of the cavity and the ribs) are populated with values. The attributes of the idealized `channel_fitting` do not have value, and therefore they will be calculated from the values of the product attributes using the relations defined in the APM.

```
DATA;

INSTANCE_OF inboard_beam;
  leg_1.cavity_3.inner_width : 2.13;
  leg_1.cavity_3.inner_breadth : 1.9345;
  leg_1.cavity_3.inner_height : 2.5932;
  leg_1.cavity_3.minimum_base_thickness : 0.50;
  leg_1.cavity_3.top_thickness : 0.45;
  leg_1.cavity_3.bottom_thickness : 0.307;
  leg_1.cavity_3.hole_diameter : 0.875;
  leg_1.cavity_3.hole_height : 0.96;
  leg_1.rib_8.thickness : 0.31;
  leg_1.rib_9.thickness : 0.31;
  leg_1.bulkhead_attach_point.end_pad.width : ?;
  leg_1.bulkhead_attach_point.end_pad.height : ?;
  leg_1.bulkhead_attach_point.end_pad.thickness : ?;
  leg_1.bulkhead_attach_point.end_pad.hole_diameter : ?;
  leg_1.bulkhead_attach_point.end_pad.hole_center_height : ?;
  leg_1.bulkhead_attach_point.base.width : ?;
  leg_1.bulkhead_attach_point.base.height : ?;
  leg_1.bulkhead_attach_point.base.thickness : ?;
  leg_1.bulkhead_attach_point.base.hole_diameter : ?;
  leg_1.bulkhead_attach_point.base.hole_center_height : ?;
  leg_1.bulkhead_attach_point.wall.width : ?;
  leg_1.bulkhead_attach_point.wall.height : ?;
  leg_1.bulkhead_attach_point.wall.thickness : ?;
END_INSTANCE;

END_DATA;
```

Figure 83-39: Inboard Beam Instances (APM-I Format)

Figure 83-40 shows the output (with values) generated by the APM Browser when the data file of Figure 83-39 is used. In this output, all the attributes – including the idealized ones that did not have value in the data file of Figure 83-39 – are shown with value.

```
inboard_beam (
    leg_1 = leg (
        cavity_3 = cavity_with_bottom_hole (
            inner_width = 2.13
            inner_breadth = 1.9345
            inner_height = 2.5932
            minimum_base_thickness = 0.5
            top_thickness = 0.45
            bottom_thickness = 0.307
            hole_diameter = 0.875
            hole_height = 0.96 )
        rib_8 = rib (
            thickness = 0.31 )
        rib_9 = rib (
            thickness = 0.31 )
        bulkhead_attach_point = channel_fitting (
            end_pad = wall_with_hole (
                width = 2.439999999999
                height = 2.088
                thickness = 0.5
                hole_diameter = 0.875
                hole_center_height = 1.266999999999 )
            base = wall_with_hole (
                width = 2.439999999999
                height = 2.843199999999
                thickness = 0.306999999999
                hole_diameter = 0
                hole_center_height = 0 )
            wall = wall (
                width = 2.843199999999
                height = 2.088
                thickness = 0.31 ) ) ))
```

Figure 83-40: APM Browser Output for the Inboard Beam

An APM client application (different from the APM Browser, which just *displays* these values) could actually *use* the values of the attributes of the idealized channel fitting to populate the channel fitting analysis model mentioned above, run the analysis, and obtain and display the results of the analysis. Such an application would use operations from the APM Protocol in order to access and manipulate APM-defined data. For example, Figure 83-41 is a constraint schematics showing how the Inboard Beam APM was used by a Channel Fitting Static Strength Analysis CBAM (Context-Based Analysis Model – see Subsection 113) developed for the PSI project. In this test case, the CBAM is implemented as an APM client application which replaces the electronic analysis template discussed above (Figure 83-36), and uses an APM to extract, transform and idealize design values. The figure illustrates how various APM attributes are connected to the analysis attributes of this CBAM, as well as some analysis results (displayed as margins of safety - MSs).

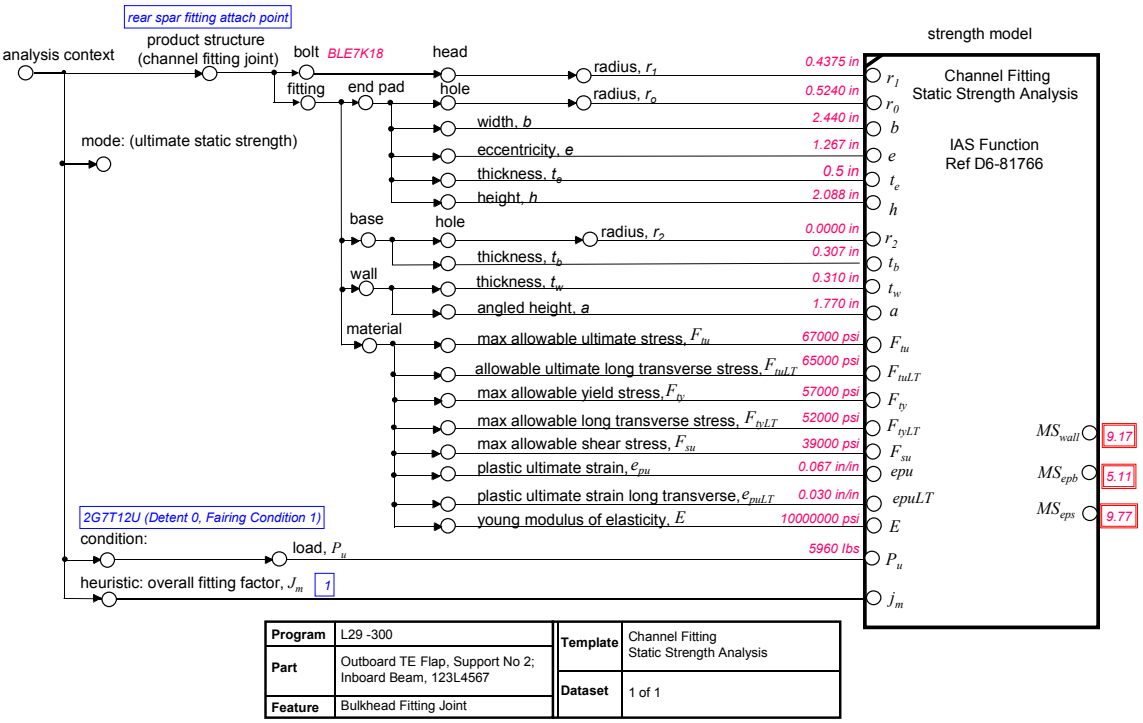


Figure 83-41: Inboard Beam APM Usage by Channel Fitting Analysis CBAM

Printed Wiring Assembly APM

The Printed Wiring Assembly APM is the most complex of the test APM definitions presented so far. The relative complexity of this APM is due to a large number of domains, significant use of aggregate attributes, and more difficult relations involving these aggregate attributes and that yield multiple solutions.

The Printed Wiring Assembly APM is the result of updating the model developed for the Tiger project (formerly known as AOPM – see Chapter 7 - and (Tamburini, Peak et al. 1996; Tamburini, Peak et al. 1997)) using the newer APM approach developed after the project was completed. The purpose of this update, besides having an additional test case to evaluate the APM approach, was to be able to compare the early-bound approach used in Tiger (see Subsection 11) with the late-bound approach of this thesis (discussed in Section 41).

The Tiger AOPM was originally conceived to support a variety of PWA and PWB analyses. These analyses included:

1. Printed Wiring Board (PWB) warpage analysis;
2. Solder joint deformation analysis;
3. Printed Wiring Assembly (PWA) warpage analysis; and
4. Plated-Through Hole (PTH) deformation analysis.

For this purpose, the Tiger AOPM contained the following PWA information (much of which can be mapped from an AP210 file):

1. PWB: its outline, its overall dimensions, detailed layup.
2. Electrical components on the board: their location, packaging geometry and material, how they are connected to the board, geometry and material of the leads and solder joints.
3. Board passages (plated-through holes, vias): their geometry, location on the board, plating material.
4. Layers that make up the layup of the board: their nominal thicknesses, material properties, function.

All the domains originally defined in the Tiger AOPM (using EXPRESS) were translated into APM-S in order to obtain an updated version of the analyzable model compatible with the newer APM approach presented in this thesis. However, for practical reasons, the focus of this updated version was on supporting the PWB bending analysis only, and therefore only the relations needed by this analysis were defined. As a result, the new APM contains more domains than those needed by the PWB bending analysis (for example, information about the electrical components and their location on the board is not used in this analysis), but having them defined anyway facilitates future extensions of the model to support other analyses. If the model is extended in the future to support other analyses, the work required will mostly involve adding new attributes and new relations to the already existing domains.

With this in mind, the information needed to support the PWB bending analysis (a subset of the information contained in the entire APM) is:

1. Detailed layup of the PWB (layers and their materials, nominal thicknesses, etc.).

2. Layer functions (signal, ground, plane, etc.). As it will be explained later, this is important to calculate the post-lamination thickness of the board.
3. Outline (geometry) of the board.

In addition to this product information, the following five idealized attributes of the PWB are needed to perform the PWB bending analysis:

1. Its coefficient of thermal bending ($\sim_B$);
2. Its width;
3. Its length;
4. Its total diagonal length; and
5. Its nested thickness (also known as post-lamination thickness).

The first of these idealized attributes - the coefficient of thermal bending ($\sim_B$) - can be thought as a lumped coefficient of thermal expansion of the total layup that reflects the degree of non-symmetry that could promote board warpage. Basically, it is calculated as the weighted sum of the coefficients of thermal expansion of the individual layers. The formula used to calculate the coefficient of thermal bending is (Tamburini, Peak et al. 1996; Tamburini, Peak et al. 1997):[52]

$$\alpha_B = C_1 \frac{\sum_1^n t_i \alpha_i y_i}{\left(t^2/2\right)} + C_2 \frac{\sum_1^n \left|t_i \alpha_i y_i\right|}{\left(t^2/2\right)} + C_3 \qquad \text{(Equation 1)}$$

Where:

α_B : coefficient of thermal bending of the board (in $^oF^{-1}$);

t_i : nominal thickness of the i^{th} layer of the layup;

[52] This formula was informally derived for the Tiger project for demonstration purposes only and has not been properly validated. The values for C_1, C_2 and C_3 were established by fitting them to limited experimental measurements of warpage from (Yeh 1992) and (Stiteler 1996). Hence, this formula should not be considered generally valid and therefore should not be used when an accurate prediction of the value of $\sim_B$ is needed. For the purposes of this test case, however, the validity of the formula was not the main issue. The assumption is that this equation may be easily replaced later with

α_i : coefficient of thermal expansion of the i^{th} layer of the layup (in $^oF^{-1}$);

y_i : distance from the i^{th} layer to the center of the board;

t : total pre-laminated thickness of the board;

n : number of layers in the layup;

$C_1 = 653.3$;

$C_2 = C_1/100000 = 0.006533$;

$C_3 = 3.496038e\text{-}7$ (in $^oC^{-1}$).

In the formula above, the distance from the layer to the center of the board times the thickness of the layer is considered to be the weight used to calculate the weighted average. Hence, thicker layers will have a stronger influence on the value of α_B than thinner layers, and layers farther away from the center will have a stronger influence than layers closer to the center. When the layup is symmetrical, the first term in the right-hand side of Equation 1 is equal to zero, because the summation terms corresponding to the layers above the center will cancel out with the terms corresponding to the layers below the center. The second term of the equation, however, is not zero, because absolute values are used instead and therefore the terms in the summation do not cancel out. When the layup is not symmetrical, then the first term has some non-zero value. The effect of this lack of symmetry on the final value of $\sim_B$ will be considerable, since the factor to which the first term is multiplied (C_1) is ten thousand times the factor to which the second term is multiplied (C_2). This is consistent with the fact that non-symmetrical layups will normally be subjected to much larger deformations that their symmetrical counterparts (hence the reason why they are usually avoided).

The second and third idealized attributes needed to perform the PWB bending analysis - the total width and length of the board - are assumed to be the width and length of the smallest imaginary rectangle enclosing the outline of the board. The formulas used to calculate them are:

$$\text{width} = \text{MAX}(x_i) - \text{MIN}(x_i) \qquad \text{(Equation 2)}$$

---

a more accurate one when it becomes available. Moreover, it is very likely that a formal equation will involve approximately the same attributes and operations of Equation 1.

$$\text{length} = \text{MAX}(y_i) - \text{MIN}(y_i) \qquad \text{(Equation 3)}$$

Where:

x_i: x coordinate of the i^{th} point of the outline of the board;

y_i: y coordinate of the i^{th} point of the outline of the board.

The fourth idealized attribute - the total diagonal of the board - is simply the diagonal length of the imaginary rectangle used to calculate the width and the length of the board. Therefore:

$$\text{total\_diagonal}^2 = \text{width}^2 + \text{length}^2 \qquad \text{(Equation 4)}$$

The fifth and last idealized attribute - the nested thickness of the board - is the resulting thickness after the board is heated and subjected to pressure during the lamination process. Naturally, this nested thickness is smaller than the sum of the nominal thicknesses of the individual layers that make up the layup of the board. The calculation of the nested thickness takes into account the fact that, during the lamination process, the epoxy material flows between the spaces existing between the traces of the conductive layers, thus reducing the thickness. The nested thickness of the board is the sum of the individual nested thicknesses of each layer, as stated by the following equation:

$$\text{board\_nested\_thickness} = \sum_{1}^{n} \text{nested\_thickess}_i \qquad \text{(Equation 5)}$$

Where:

board_nested_thickness : nested thickness of the board;

$\text{nested\_thickness}_i$: nested thickness of the i^{th} layer of the layup (calculated with Equations 6, 7, or 9 below);

n : number of layers in the layup.

The nested thicknesses of each layer are calculated differently depending on whether the layer is a copper foil, a prepreg set or a copper-cladded laminate. For copper layers it is assumed that the thickness remains constant during lamination. Therefore the nested thickness for copper layers is equal to:

$$\text{nested\_thickness}_{\text{copper}} = \text{nominal\_thickness} \quad \text{(Equation 6)}$$

Where:

$\text{nested\_thickness}_{\text{copper}}$: nested thickness of a copper layer;

nominal_thickness : nominal thickness of the copper foil.

For prepreg sets, the nested thickness is calculated using the following equation:

$$\text{nested\_thickness}_{\text{prepreg\_set}} = \sum_{1}^{p} 0.9 \times \text{ho}_i - \text{resin\_to\_fill} \quad \text{(Equation 7)}$$

$$\text{resin\_to\_fill} = \text{nominal\_thickness}_{\text{bottom}} \times \text{percent\_etched}_{\text{bottom}} + \text{nominal\_thickness}_{\text{top}} \times \text{percent\_etched}_{\text{top}} \quad \text{(Equation 8)}$$

Where:

$\text{nested\_thickness}_{\text{prepreg\_set}}$: nested thickness of a prepreg set;

ho_i : height of the i^{th} prepreg sheet of the prepreg set;

resin_to_fill : thickness of resin to fill;

$\text{nominal\_thickness}_{\text{bottom}}$: nominal thickness of the copper foil below the prepreg set;

$\text{nominal\_thickness}_{\text{top}}$: nominal thickness of the copper foil above the prepreg set;

$\text{percent\_etched}_{\text{bottom}}$: percent of copper etched from the copper foil below the prepreg set;

$\text{percent\_etched}_{\text{top}}$: percent of copper etched from the copper foil above the prepreg set;

p : number of prepreg sheets in the prepreg set.

Equation 7 gives the nested thickness of a prepreg set by adding the post-lamination thicknesses of the individual prepreg sheets that make up the set (assumed to be 90% of their original thicknesses, ho) and then subtracting the thickness of resin to fill. The thickness of resin to fill is calculated with Equation 8, which takes into account the flow of epoxy into the void spaces left between the traces of etched copper. As the two equations imply, the more copper etched out of a copper foil, the more resin to fill and therefore the smaller the resulting nested thickness. For example, a *signal* copper layer (which in general has many traces and therefore a larger percentage of copper etched) will have more void

space to fill than a *ground* layer (that has no traces and therefore no copper etched). The exact amount of copper etched from a copper layer depends on the specific design of the circuit board. However, it is impractical to accurately calculate this percentage for all layers in all boards and therefore *average* percentages of etched copper are used instead, considering the function of the layer (signal, power, solder, etc.) as a guideline. The following percentages are commonly used in industry (Lewis 1996):

For signal layers: 70%

For power layers: 30%

For solder layers: 0%

For mixed layers: 50%

For plane layers: 30%

For ground layers: 30%

For layers with components: 0%

For any other type of layer: 30%

Finally, the nested thickness for copper-cladded laminates is simply the sum of the nominal thicknesses of the laminate core and the top and bottom copper foils:

$$nested\_thickness_{laminate} = nominal\_thickness_{core} + nominal\_thickness_{top} + nominal\_thickness_{bottom} \quad \text{(Equation 9)}$$

Where:

$nested\_thickness_{laminate}$: nested thickness of a copper-cladded laminate;

$nominal\_thickness_{core}$: nominal thickness of the core of the copper-cladded laminate;

$nominal\_thickness_{top}$: nominal thickness of the top copper foil of the laminate;

$nominal\_thickness_{bottom}$: nominal thickness of the bottom copper foil of the laminate.

The complete definition of the Printed Wiring Assembly APM is too long to be included here and therefore is included in Appendix Z.1. Only those portions relevant to the PWB bending analysis will be reproduced and discussed here.

The Printed Wiring Assembly APM contains two source sets: `pwa_geometry` and `pwb_layup`. Source set `pwa_geometry` contains domains to define, among other things, the outline and overall dimensions of the board, the components and their locations, passages (such as vias and plated-through holes), and how components are connected to the board (with leads, ball grid arrays, directly to the board). From this source set, only domain `pwb` is of interest for the PWB bending analysis. Domain `pwb` is defined as follows (not all of its attributes are listed):

```
DOMAIN pwb SUBTYPE_OF multimaterial_part;
   IDEALIZED total_diagonal : REAL;
   IDEALIZED width : REAL;
   IDEALIZED length : REAL;
   outline : LIST[1,?] OF xy_coordinates;
   layup : STRING;
   pre_lamination_thickness : REAL;
   IDEALIZED nested_thickness : REAL;
   IDEALIZED coefficient_of_thermal_bending : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
   pir1: "<coefficient_of_thermal_bending>*
         (<pre_lamination_thickness>^2) ==
         (2*653.3*<layup.layers.SUM[t_times_cte_times_y]>) +
         (2*0.006533*<layup.layers.SUM[t_times_cte_times_abs_y]>) +
          3.496038*^-7*(<pre_lamination_thickness>^2)";
   pir2: "<width> == <outline.MAX[x]> - <outline.MIN[x]> ";
   pir3: "<length> == <outline.MAX[y]> - <outline.MIN[y]> ";
   pir4: "<total_diagonal>^2 == <length>^2 + <width>^2";
   pir5: "<nested_thickness> == <layup.nested_thickness>";
   pir6: "<pre_lamination_thickness> ==
          <layup.layers.SUM[pre_lamination_thickness]>";
END_DOMAIN;
```

Six product idealization relations are defined in domain `pwb` above. Relations `pir1` to `pir4` correspond directly to Equations 1 to 4. In relation `pir1`, the term:

```
layup.layers.SUM[t_times_cte_times_y]
```
[53]

corresponds to the numerator of the first term of the right-hand side of Equation 1:

---

[53] This way to express the argument for the `SUM` function (`t_times_cte_times_y`) is due to a current limitation of the implementation of the `SUM` function that limits its arguments to just one real number. As a consequence, the auxiliary variable `t_times_cte_times_y` has to be used in lieu of the operation t*cte*y. This auxiliary variable – and the operation to calculate its value – must be defined in each subtype of `pwb_layer`, as presented in a few paragraphs.

$$\sum_{1}^{n} t_i \alpha_i y_i$$

and the term:

```
layup.layers.SUM[t_times_cte_times_abs_y]
```

corresponds to the numerator of the second term of the right-hand side of Equation 1:

$$\sum_{1}^{n} \left| t_i \alpha_i y_i \right|$$

Relation `pir5` simply equates the nested thickness of the board and the nested thickness of its layup (which, in turn, is the sum of the nested thicknesses of the individual layers, as it will be explained below). Relation `pir6` states that the pre-lamination thickness of the board is the sum of the pre-lamination thicknesses of its individual layers.

The second source set of this APM (`pwb_layup`) contains entities to define the detailed layup of the board. The root domain of this source, domain `layup`, is defined as follows:

```
DOMAIN layup;
   pwb_part_number : STRING;
   layers : LIST[1,?] OF pwb_layer;
   IDEALIZED nested_thickness : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
   pir7: "<nested_thickness> == <layers.SUM[nested_thickness]>";
END_DOMAIN;
```

This domain defines the detailed layup of a PWB. Instances of `layup` from this source set are joined with instances of `pwb` from the first source set when the value of attribute `layup` of domain `pwb` is equal to the value of attribute `pwb_part_number` of domain `layup`. This is defined by the source set link shown below:

```
pwa_geometry.pwa.associated_pwb.layup ==
   pwb_layup.layup.pwb_part_number;
```

As the result of this link, attribute `layup` of `pwb` will point to an instance of domain `layup`.

The individual layers of the layup are contained in attribute `layers` of domain `layup`. This attribute is a list whose elements are of type `pwb_layer`. Relation `pir7` of domain `layup`

states that the nested thickness of the layup is equal to the sum of the nested thicknesses of the individual layers (as stated by Equation 5).

Domain pwb_layer is defined as follows:

```
DOMAIN pwb_layer;
   description : STRING;
   total_length : REAL;
   total_width : REAL;
   total_height : REAL;
   primary_structural_material : solid_material;
   IDEALIZED nested_thickness : REAL;
   pre_lamination_thickness : REAL;
   distance_from_board_symmetry_axis : REAL;
   abs_distance_from_board_symmetry_axis : REAL;
   t_times_cte_times_y : REAL;
   t_times_cte_times_abs_y : REAL;
END_DOMAIN;
```

An important aspect of the APM representation highlighted by this test case should be discussed at this point. Domain pwb_layer has three subtypes: pwb_copper_foil, pwb_prepreg_set, and pwb_copper_cladded_laminate. As a result, since attribute layers in domain layup is a list of pwb_layers the elements of this list may be instances of any of these three domains. For example, the first layer of this list may be a copper layer, the second a prepreg set, the third a laminate, and so on.

These three domains share some common attributes (defined in their common supertype pwb_layer) such as nested_thickness, pre_lamination_thickness, t_times_cte_-times_y, and t_times_cte_times_abs_y, among others. However, the relations in which these attributes participate are different in each of these three domains. For example, in domain pwb_copper_foil:

```
DOMAIN pwb_copper_foil SUBTYPE_OF pwb_layer;
   weight_per_unit_area : REAL;
   layer_function : STRING;
   min_thickness : REAL ;
   nominal_thickness : REAL ;
   max_thickness : REAL ;
   percent_etched : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
   pir8: "<nested_thickness> == <nominal_thickness>";
   pir9: "<pre_lamination_thickness> == <nominal_thickness>";
```

```
    pir10: "<t_times_cte_times_y> ==
            <pre_lamination_thickness>*
            <primary_structural_material.
                associated_linear_elastic_model.cte>*
            <distance_from_board_symmetry_axis>";
   pir11: "<t_times_cte_times_abs_y> ==
           <pre_lamination_thickness>*
           <primary_structural_material.
               associated_linear_elastic_model.cte>*
           <abs_distance_from_board_symmetry_axis>";
END_DOMAIN;
```

relation pir8 states that nested_thickness is equal to nominal_thickness (as stated by Equation 6), whereas in domain pwb_prepreg_set:

```
DOMAIN pwb_prepreg_set SUBTYPE_OF pwb_layer;
   prepregs : LIST[1,?] OF pwb_prepreg_sheet;
   top_copper_layer : pwb_copper_foil;
   bottom_copper_layer : pwb_copper_foil;
 PRODUCT_IDEALIZATION_RELATIONS
   pir12: "<nested_thickness> == <prepregs.SUM[ho]>*0.9 -
            <top_copper_layer.nominal_thickness>*
            <top_copper_layer.percent_etched> -
            <bottom_copper_layer.nominal_thickness>*
            <bottom_copper_layer.percent_etched>";
   pir13: "<pre_lamination_thickness> ==
           <prepregs.SUM[nominal_thickness]>";
   pir14: "<t_times_cte_times_y> ==
           <pre_lamination_thickness>*
           <primary_structural_material.
               associated_linear_elastic_model.cte>*
           <distance_from_board_symmetry_axis>";
   pir15: "<t_times_cte_times_abs_y> ==
           <pre_lamination_thickness>*
           <primary_structural_material.
               associated_linear_elastic_model.cte>*
           <abs_distance_from_board_symmetry_axis>";
END_DOMAIN;
```

relation pir12 states a different relation (corresponding to Equation 7) to calculate nested_thickness. Finally, in domain pwb_copper_cladded_laminate:

```
DOMAIN pwb_copper_cladded_laminate SUBTYPE_OF pwb_layer;
   related_core : pwb_core;
   laminate_id : STRING;
   top_copper_layer : pwb_copper_foil;
```

```
    bottom_copper_layer : pwb_copper_foil;
  PRODUCT_IDEALIZATION_RELATIONS
    pir16: "<nested_thickness> == <related_core.nominal_thickness> +
            <top_copper_layer.nominal_thickness> +
            <bottom_copper_layer.nominal_thickness>";
    pir17: "<pre_lamination_thickness> ==
            <related_core.nominal_thickness> +
            <top_copper_layer.nominal_thickness> +
            <bottom_copper_layer.nominal_thickness>";
    pir18: "<t_times_cte_times_y> ==
            <related_core.nominal_thickness>*
            <primary_structural_material.
                associated_linear_elastic_model.cte>*
            <distance_from_board_symmetry_axis> +
            <top_copper_layer.t_times_cte_times_y> +
            <bottom_copper_layer.t_times_cte_times_y>";
    pir19: "<t_times_cte_times_abs_y> ==
            <related_core.nominal_thickness>*
            <primary_structural_material.
                associated_linear_elastic_model.cte>*
            <abs_distance_from_board_symmetry_axis> +
            <top_copper_layer.t_times_cte_times_abs_y> +
            <bottom_copper_layer.t_times_cte_times_abs_y>";
  END_DOMAIN;
```

relation `pir16` defines yet another relation to calculate `nested_thickness` (this one corresponding to Equation 9). A similar situation occurs with the relations involving attributes `pre_lamination_thickness` (relations `pir9`, `pir13`, and `pir17`), `t_times_cte_times_y` (relations `pir10`, `pir14`, and `pir18`), and `t_times_cte_times_abs_y` (relations `pir11`, `pir15`, and `pir19`).

It is important to point out that a relation does not have to be repeated in every subtype of a given domain if it is identical in each subtype. When this is the case, the relation can be defined only once in the parent domain and inherited by each of the subtypes. Moreover, it should be possible to *override* a relation in any of the subtypes of a given domain. With this in mind, a closer look at relations `pir10` and `pir14` (in domain `pwb_copper_foil` and `pwb_prepreg_set`, respectively) will reveal the fact that these two relations are mathematically identical and therefore they were unnecessarily duplicated in domains `pwb_copper_foil` and `pwb_prepreg_set`. Just one of them (say `pir10`) could have been defined in their common supertype (`pwb_layer`) and the three subtypes of `pwb_layer` would have inherited the relation from it. Next, since `t_times_cte_times_y` is calculated

differently in the case of laminates, `pir10` could have been overridden by defining relation `pir18` in domain `pwb_copper_cladded_laminate`. At the end, only *two* relations would have been defined instead of *three* as done above. A similar reasoning applies for relations `pir11` and `pir15`. *The only reason this is not done in the examples above is because the prototype implementation presented in this thesis does not support overriding relations*. As it will be discussed in Chapter 110, support for this feature should be added in future implementations of the APM Protocol.

Two STEP P21 data files (each corresponding to one source set of the APM) were used to test this APM and are included in Appendix DD.1. They were originally created for the Tiger project and required only minor changes to be able to run this test case with the prototype implementation of this thesis. Much of the data corresponding to the `pwa_geometry` source set had been originally created with an E/CAD tool (Mentor Graphics) in STEP AP210 format and translated into APM format with a mapping program developed by the author using STEP Tools Inc.'s ST-Developer Toolkit (STEP Tools Inc 1997b; STEP Tools Inc 1997c).

A portion of the output created by the APM Browser showing the resulting values for total diagonal, width, length, pre-lamination thickness, nested thickness, $\sim_B$ (in bold) is shown below:

```
pwb (
   part_number = "BF906_1006-PWB"
   total_diagonal = 5.445181356024
   width = 3.799999999999
   length = 3.9
   outline = ListOfxy_coordinatess (
                    xy_coordinates = xy_coordinates (
                          x = 0
                          y = 0.45 )
                    (... other coordinates ) )
   layup = layup (
      pwb_part_number = "BF906_1006-PWB"
      layers = ListOfpwb_layers (
         pwb_copper_foil = pwb_copper_foil (
            description = "Layer 1"
            ... other attributes ) )
         pwb_prepreg_set = pwb_prepreg_set (
            description = "Prepreg set 1"
            ... other attributes ) )
```

```
        pwb_copper_cladded_laminate = pwb_copper_cladded_laminate (
           description = "Layers 2 and 3"
           ... other attributes ) )
        pwb_prepreg_set = pwb_prepreg_set (
           description = "Prepreg set 2"
           ... other attributes ) )
        pwb_copper_cladded_laminate = pwb_copper_cladded_laminate (
           description = "Layers 4 and 5"
           ... other attributes ) )
        pwb_prepreg_set = pwb_prepreg_set (
           description = "Prepreg set 3"
           ... other attributes ) )
        pwb_copper_foil = pwb_copper_foil (
            description = "Layer 6"
           ... other attributes ) )
pre_lamination_thickness = 0.067399999999
nested_thickness = 0.058099999999
coefficient_of_thermal_bending = 0.000000385897 )
```

Besides the APM Browser (whose output is shown above), this APM is also used by the PWB Bending Analysis Application (presented in Subsection 90) to calculate the deflection of a PWB when it is subjected to a temperature change.

Test APM Client Applications

In essence, APM client applications are programs that utilize the classes and operations - provided in the APM class library - to access information defined by an APM. The APM Applications described in this section provide concrete examples of how the operations presented in Subsection 77 can be put into use. In general, APM client applications take advantage of the analysis-oriented view of a part or product provided by the APM to perform some form of engineering analysis. These applications normally feature a graphical user interface and, in some cases, communicate with external solution engines or other programs.

The following four APM client applications were developed for this thesis (and will be presented in order of complexity):

1. PWB Bending Analysis Application (Subsection 90);

2. Flap Link Extensional Analysis Application (Subsection 91);
3. Back Plate Analysis and Synthesis Application (Subsection 92); and
4. The APM Browser (Subsection 93).

APM Client applications may or may not be tied to a particular APM definition. APM applications tied to a particular APM are called *APM-Specific applications*, while those not tied to a particular APM are called *generic applications*. Of the applications listed above, one is an APM generic application (the APM Browser) and the rest are APM-Specific applications (the PWB Bending Analysis Application is tied to the Printed Wiring Assembly APM presented in Subsection 88, the Flap Link Extensional Analysis Application to the Flap Link APM presented in Subsection 85, and the Back Plate Analysis and Synthesis Application to the Back Plate APM presented in Subsection 86).

These four APM client applications are written in Java and use the classes and methods provided by the implementation of the APM Representation presented in Sections 76 and 77. Since these classes provide much of the functionality required for design-analysis integration (such as support for multi-fidelity idealizations, multi-directional constraint solving, design data linking, etc.), the code of these client applications is relatively simple.

In the subsections that follow, only specific portions of the code of these APM applications will be presented and discussed. The focus will be on those sections of the code in which an important APM operation is being performed. The code that creates the graphical-user interfaces and handles the user interactions will not be explained, since these aspects are considered to be out of scope for the purposes of this discussion. For the interested reader, the complete code of these applications is included in Appendix II.

PWB Bending Analysis Application

The PWB Bending Analysis Application is a simple analysis application whose purpose is to calculate the deflection (or warpage) undergone by a PWB when subjected to a uniform change in temperature. This application utilizes the Printed Wiring Assembly APM presented in Subsection 88. It is interesting to point out that even though this application uses the most complex of the APMs presented in Section 84, it is actually the simplest of the APM client applications presented in this thesis. This is an indication of how an APM

definition - in combination with the functionality provided by the operations defined in the APM protocol - hide most of the complexities involved in the creation of an analyzable view of a product.

In order to calculate the warpage undergone by a PWB due to an uniform change in temperature, this application uses the following formula (resulting from modeling the PWB as a simple layered beam, see Gieck, Kurt et al. 1990; Guyer 1989):

$$\delta = \frac{\alpha_B \times D^2 \times \Delta T}{t} \qquad \text{(Equation 10)}$$

Where:

δ : maximum deflection at the edge of the board;

α_B : coefficient of thermal bending of the board;

D : maximum diagonal length of the board. Assumed to be equal to the diagonal of a rectangle with dimensions W x L, where W is the width of the board and L is the length of the board;

ΔT : temperature change;

t : nested (post-lamination) thickness of the board.

All the terms involved in this formula (with the exception of ΔT and δ) are *idealized* attributes of the PWB and are defined as such in the APM definition (that is, they are preceded with the APM-S keyword IDEALIZED, see Appendix Z.1 or Subsection 88). ΔT and δ are input and the output of this analysis, and therefore are not part of the product definition (that is, they are not defined in the APM). Figure 83-42 shows the PWB with these parameters.

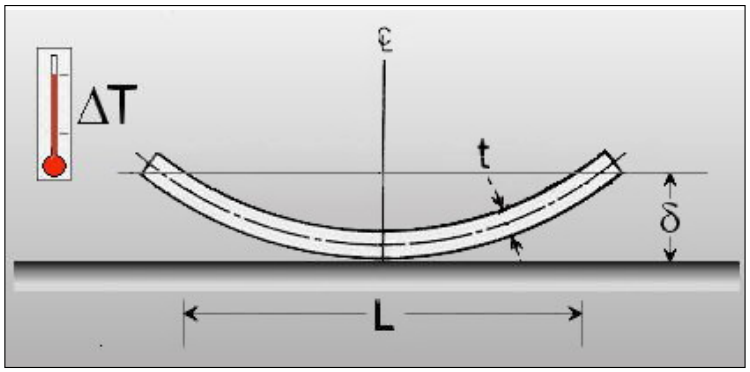


Figure 83-42: PWB Bending Analysis Model

Figure 83-43 is a screen shot of the first screen that appears when this application is started. The first action the user must perform is to load the APM definition by selecting the file where the APM-S definition of the APM is stored[54]. This is a necessary step, because the application needs to know the domains, attributes, relations and source set links defined in the analyzable product model.

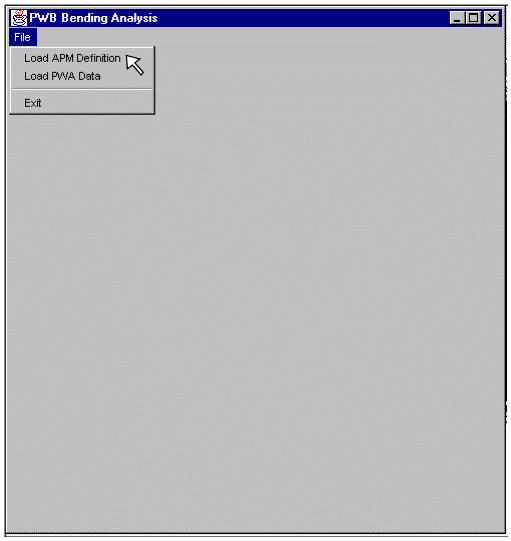


Figure 83-43: PWB Bending Analysis Application (Loading the APM)

[54] Alternatively, since this application will always use the same APM, the application could load the APM definition automatically at start-up time.

Once the user selects the APM definition file, method **APMInterface.-loadAPMDefinitions** is used to load the APM definition into memory as follows:

```
APMInterface.loadAPMDefinitions( apmDefinitionFileName );
```

where **apmDefinitionFileName** is the name of the APM definition file selected by the user. As discussed in Subsection 78, method **APMInterface.loadAPMDefinitions** parses the APM definition file (a text file written in APM-S format), creates the corresponding APM instances in memory (instances of **APMSourceSet**, **APMDomain**, **APMAttribute**, **APMRelation**, **ConstraintNetwork**, **ConstraintNetworkNode**, **APMSourceSetLink**), and proceeds to link the domain definitions as specified by the source set links defined in the APM (see the domain linking method **linkAPMDefinitions** in Subsection 78. All this activity is hidden from the developer of the application and packaged into one single operation (**loadAPMDefinitions**).

After the APM definitions are loaded, the next step is to load the design data required to run the analysis. Recall from the definition of source set in Subsection 45 that there must be one data file for each source set defined in the APM. The number of source sets that are defined in a given APM can be found with the following method:

```
int numberOfSourceSets = APMInterface.getSourceSets().size();
```

In this case, finding the number of source sets is really not necessary because this application will always use the Printed Wiring Assembly APM, and therefore the number of source sets is known in advance (this APM has *two* source sets: `pwa_geometry` and `pwb_layup`). Therefore, the user will be prompted for two data files. These data files are the repositories where instances of the domains of each source set defined in the APM are stored. The first file corresponds to the repository where the PWA geometry data is stored (as instances of domains defined in source set `pwa_geometry`), and the second file corresponds to the repository where the detailed layup of the PWB is stored (as instances of domains defined in source set `pwb_layup`)[55]. In the prototype implementation presented in this thesis, the data in these files may be either in APM-I or in STEP P21 format. Support for additional data

[55] These two STEP P21 files were originally created for the Tiger project (Peak, Fulton et al. 1997; Tamburini, Peak et al. 1996; Tamburini, Peak et al. 1997). The file containing the PWA geometry was obtained by translating a STEP AP210 file created with an E/CAD tool (Mentor Graphics) using a translation program written by the author with STEP Tools Inc.'s ST-Developer toolkit (STEP Tools Inc 1997c). The second file was generated by a PWB layup tool application, also developed by the author.

formats could be added in future implementations by adding a new subtype to class **APMSourceDataWrapper** for each new format to be supported (see Subsection 79).

The application stores the names of the two files selected by the user in a list of strings (for example, **listOfFileNames**). Next, the following method:

```
APMInterface.loadSourceSetData( listOfFileNames );
```

loads the data from the source files, creates the corresponding APM instances in memory (instances of **APMDomainInstance**), and proceeds to link them according to what is specified by the source set links defined in the APM (see the data linking method **linkSourceSetData** in Subsection 79). It is important to point out that the developer of this application does not have to write any code to handle the different formats in which the data may be stored. These details are handled by method **loadSourceSetData**. Therefore, if the format of any of the source data files should change, the code to load the data remains unchanged.

Once the data is loaded into memory and linked, the developer will normally use APM Protocol operations to access and manipulate it. The next step in this application is to get all the instances of domain pwa that were read from the source files. This is performed with method **APMInterface.getInstancesOf** as follows:

```
ListOfAPMComplexDomainInstances listOfPWAInstances =
   APMInterface.getInstancesOf( "pwa" );
```

where variable **listOfPWAInstances** is declared as a **ListOfAPMComplexDomain-Instances** because method **getInstancesOf** returns a **ListOfAPMComplexDomain-Instances**. However, since pwa is actually an APM Object Domain, the resulting list will only contain instances of **APMObjectDomainInstance** (recall from the APM Information Model – Subsection 66 - that **APMObjectDomain** is a subtype of the abstract class **APMComplexDomain**).

In this example, it is assumed that the data file contains only *one* instance of pwa. This instance will get stored as the first (and only) element of **listOfPwaInstances** and therefore can be retrieved as follows:

```
APMObjectDomainInstance pwaInstance = (APMObjectDomainInstance)
   listOfPWAInstances.elementAt( 0 );
```

Next, the application proceeds to query the values of several attributes of **pwaInstance** in order to display them on the screen and use them later in Equation 10 to calculate the deflection of the board. *It is perhaps at this point where the benefits of the APM approach become most apparent*: these values are queried from the application code in the same way – regardless of whether they come directly from the source data files or have to be calculated at run time using one or more of the relations defined in the APM. To illustrate this, consider the following code used to query the values displayed in the screen shot of Figure 83-44:

```
String pwaDescription = pwaInstance.
      getStringInstance( "description" ).
      getStringValue();

String pwaPartNumber = pwaInstance.
      getStringInstance( "part_number" ).
      getStringValue();

String pwbPartNumber = pwaInstance.
      getObjectInstance( "associated_pwb" ).
      getStringInstance( "part_number" ).
      getStringValue();

double pwbNestedThickness = pwaInstance.
      getObjectInstance( "associated_pwb" ).
      getRealInstance( "nested_thickness" ).
      getRealValue();

double pwbWidth = pwaInstance.
      getObjectInstance( "associated_pwb" ).
      getRealInstance( "width" ).
      getRealValue();

double pwbLength = pwaInstance.
      getObjectInstance( "associated_pwb" ).
```

```
        getRealInstance( "length" ).
        getRealValue();

double coefficientOfThermalBending = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getRealInstance( "coefficient_of_thermal_bending" ).
        getRealValue();
```

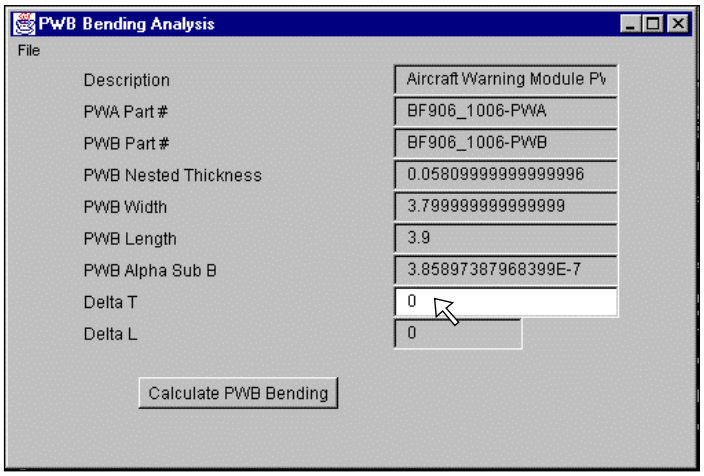


Figure 83-44: PWB Bending Analysis Application (Showing PWB Attribute Values)

The first of the statements above queries the value of attribute `description` of **`pwaInstance`** (displayed as "Description" with value "Aircraft Warning Module PWA" in Figure 83-44)[56]. This value already exists in the source data file and therefore was simply displayed without performing any calculation[57]. The same is also true for the second and third attributes (`part_number` – displayed as "PWA Part #" with value "BF906_1006-PWA", and `associated_pwb.part_number` – displayed as "PWB Part #" with value "BF906_1006-PWB"). The rest of the values correspond to idealized attributes of the PWB that need to be calculated at run time by the constraint solver using the relations defined in

[56] Alternatively, the statements above could be translated into equivalent expressions using dot notation. For example, the first statement may be interpreted as pwaDescription = pwaInstance.description, and the last statement as coefficientOfThermalBending = pwaInstance.associated_pwb.coefficient_of_thermal_bending.

[57] Conceivably, relations could be defined among string attributes as well. However, this prototype implementation does not support it.

the APM. For example, the value of `associated_pwb.nested_thickness` (displayed as "PWB Nested Thickness" with value 0.05809 in) was not originally populated in the source data and therefore had to be calculated using relations `pir2`, `pir7`, `pir8`, `pir12`, and `pir16` from the Printed Wiring Assembly APM (see the complete APM definition in Appendix Z.1 or portions of it in Subsection 88). The same is true for the values of `associated_pwb.width` (displayed as "PWB Width" with value 3.79999 in), `associated_pwb.length` (displayed as "PWB Length" with value 3.9 in), and `associated_pwb.coefficient_of_thermal_bending` (displayed as "PWB Alpha Sub B" with value 3.85897E-7 $^{o}C^{-1}$).

It is important to point out that regardless of how complex the calculation of a value is and how many relations are involved in this calculation, all values are queried using basically the same code. For example, the calculation of the coefficient of thermal bending of the PWB is considerably more complex than the calculation of the width of the board (compare relations `pir1` and `pir2` of the Printed Wiring Assembly APM). However, this additional complexity is not reflected in the code, since the code to query these two values is essentially the same. More importantly, no code is needed to decide which relations should be used in order to find the value of a particular attribute. Moreover, if for any reason the relations defined in the APM change the code of the application does not have to be modified to reflect the change, since the updated relations will automatically be used by the constraint solver at run time. As discussed in Section 81, method **`APMRealInstance.getRealValue`** handles all these constraint-solving details for the developer.

In the case of APM-Specific applications, the developer must refer to the APM to get structural information such as the names of the attributes and their domain types. For example, in order to be able to write:

```
double coefficientOfThermalBending = pwaInstance.
      getObjectInstance( "associated_pwb" ).
      getRealInstance( "coefficient_of_thermal_bending" ).
      getRealValue();
```

the developer must know that domain `pwa` (of which **`pwaInstance`** is an instance) has an attribute called "**`associated_pwb`**", whose type is an object domain with an attribute called "**`coefficient_of_thermal_bending`**", which is a real value. However, the developer does

not have to check whether the attribute has a value or not or, if the attribute does not have a value, what relations are needed to calculate it.

Once the values of the attributes of interest are obtained, the last step is to use them to perform the analysis. In this case, the values of **`coefficientOfThermalBending`**, **`pwbWidth`**, **`pwbLength`**, and **`pwbNestedThickness`** are used to calculate the deflection of the board (**`deltaL`**). First, the user must enter a value for ΔT (the only analysis input in this case). Next, Equation 10 is used to get the value of the deflection as follows:

```
double deltaL = (coefficientOfThermalBending *
      (pwbWidth * pwbWidth + pwbLength * pwbLength) * deltaT )/
      pwbNestedThickness;
```

Where:

`deltaL` : resulting deflection of the board;

`deltaT` : temperature change (entered by the user).

Figure 83-45 shows an example in which the user has entered a value for ΔT of 125 °C, resulting in a deflection of 0.02461 in.

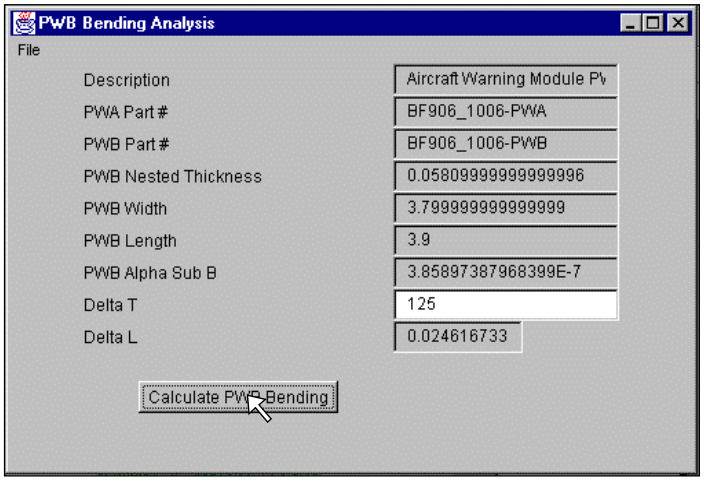


Figure 83-45: PWB Bending Analysis Application (Showing Analysis Results)

Flap Link Extension Analysis Application

The Flap Link Extensional Analysis Application was used in several occasions as an example in the previous chapter - together with the Flap Link APM presented in Subsection 85 – to introduce the APM approach and illustrate some of the APM fundamental concepts. The paragraphs that follow will describe this application in more detail.

The purpose of the Flap Link Extensional Analysis Application is to calculate the elongation of a flap link when it is subjected to an axial load. For that purpose, it uses the Flap Link APM presented in Subsection 85.

From the point of view of the APM operations used, this application is very similar to the PWB Bending Analysis application described in the previous subsection. Both applications use the same APM operations to load the APM (method **`loadAPMDefinitions`**), load the data (method **`loadSourceSetData`**), find the instances of a specific domain (method **`getInstanceOf`**), and query the values of specific attributes (methods **`getObjectInstance`**, **`getRealInstance`**, **`getRealValue`**, **`getStringInstance`**, **`getStringValue`**).

Once the APM definition and data are loaded with methods **`loadAPMDefinitions`** and **`loadSourceSetData`**, respectively, method **`getInstancesOf`** is used to get the list of flap link instances read from the source data file as follows:

```
ListOfAPMComplexDomainInstances listOfFlapLinkInstances =
   APMInterface.getInstancesOf( "flap_link" );
```

Method **`getInstancesOf`** may return more than one instance of `flap_link` from the source data file. The user selects he desired instance from a drop-down list displayed in the initial screen (see screen shot in Figure 83-46). The instance selected is stored in a separate variable (**`APMObjectDomainInstance selectedFlapLinkInstance`**).

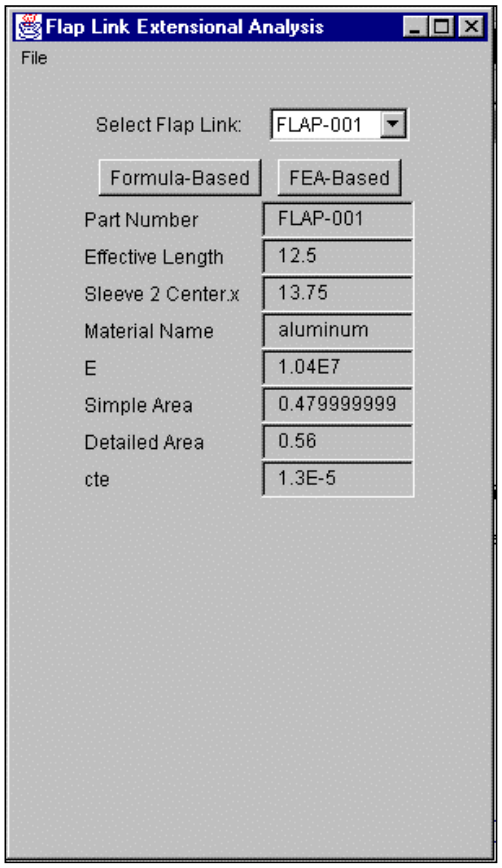


Figure 83-46: Flap Link Extensional Analysis Application (Showing Information for Flap Link "FLAP-001" Selected)

Once the user selects the desired flap link instance, the application proceeds to query the values of some of the flap link attributes (Figure 83-46) as follows:

```
String partNumber = selectedFlapLinkInstance.
      getStringInstance( "part_number" ).
      getStringValue();

double L = selectedFlapLinkInstance.
      getRealInstance( "effective_length" ).
      getRealValue();

double x2 = selectedFlapLinkInstance.
      getObjectInstance( "sleeve_2" ).
```

```
      getObjectInstance( "center" ).
      getRealInstance( "x" ).
      getRealValue();

String materialName = selectedFlapLinkInstance.
      getObjectInstance( "material" ).
      getStringInstance( "name" ).
      getStringValue();

double E = selectedFlapLinkInstance.
      getObjectInstance( "material" ).
      getMultiLevelInstance( "stress_strain_model" ).
      getObjectInstance("temperature_independent_linear_elastic").
      getRealInstance( "youngs_modulus" ).
      getRealValue( );

double cte = selectedFlapLinkInstance.
      getObjectInstance( "material" ).
      getMultiLevelInstance( "stress_strain_model" ).
      getObjectInstance("temperature_independent_linear_elastic"
      ).
      getRealInstance( "cte" ).
      getRealValue( );

double simpleA = selectedFlapLinkInstance.
      getObjectInstance( "shaft" ).
      getMultiLevelInstance( "critical_cross_section" ).
      getObjectInstance( "simple" ).
      getRealInstance( "area" ).
      getRealValue( );

double detailedA = selectedFlapLinkInstance.
      getObjectInstance( "shaft" ).
      getMultiLevelInstance( "critical_cross_section" ).
```

```
getObjectInstance( "detailed" ).
getRealInstance( "area" ).
getRealValue( );
```

Attribute `effective_length` queried above is an idealized attribute of domain `flap_link`. Again, as discussed in the previous section, no additional code is required to solve for its value since method **`getRealValue`** handles the constraint-solving details.

A new feature introduced in this application is the utilization of multi-level domains. Recall from the Flap Link APM (Subsection 85) that domain `flap_link` has an attribute called `shaft`, which in turn has an attribute called `critical_cross_section` whose type is a multi-level domain called `cross_section` with two levels (`detailed` and `simple`) as shown below:

```
DOMAIN flap_link;
   (*  other flap_link attributes *)
   shaft : beam;
END_DOMAIN;

DOMAIN beam;
   (*  other beam attributes *)
   critical_cross_section : MULTI_LEVEL cross_section;
END_DOMAIN;

MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
END_MULTI_LEVEL_DOMAIN;
```

When the *simple* version of the cross section is needed, the following query from above is performed:

```
double simpleA = selectedFlapLinkInstance.
      getObjectInstance( "shaft" ).
      getMultiLevelInstance( "critical_cross_section" ).
      getObjectInstance( "simple" ).
      getRealInstance( "area" ).
      getRealValue( );
```

whereas when the *detailed* version is needed, the following query is performed:

```
double detailedA = selectedFlapLinkInstance.
      getObjectInstance( "shaft" ).
      getMultiLevelInstance( "critical_cross_section" ).
      getObjectInstance( "detailed" ).
      getRealInstance( "area" ).
      getRealValue( );
```

The values for the simplified and detailed areas are displayed in the screen shown in Figure 83-46 above and labeled "Simple Area" and "Detailed Area", with values of 0.4799 in^2 and 0.56 in^2, respectively.

This application demonstrates how an APM can support more than one solution method and level of idealization fidelity. The first - support for multiple solution methods - is demonstrated by giving the user the option to calculate the elongation of the flap link using either a simple formula-based model or a more detailed finite-element analysis. The second – support for multiple levels of idealization fidelity – is demonstrated implicitly when the user selects the solution method. When he or she selects the formula-based model, the application utilizes a 1-D representation of the flap link, whereas when he or she selects the FEA-based model the application utilizes a 2-D representation.

When the formula-based analysis model is selected, the following formula to calculate the elongation **ΔL** of a rod subjected to an axial load **P** and a change in temperature **ΔT** is used (Gere and Timoshenko 1990):

$$\Delta L = \frac{PL}{EA} + \alpha (\Delta T) L \qquad \text{(Equation 11)}$$

Where:

αL : elongation of the flap link;

P : applied axial force;

L : effective length of the flap link;

E : Young's modulus;

A : critical cross section of the flap link (simple or detailed);

~ : coefficient of thermal expansion;

αT : temperature change.

When the formula-based solution method is selected, the user is also given the choice of using either the detailed or the simple version of the critical cross section of the flap link as the value of **A** in the formula above. This, again, demonstrates the support for multiple levels of idealization fidelity. Figure 83-47 shows the analysis results when the simple cross section is selected (Delta L = 0.00268 in and Stress-X = 208.33 psi) and Figure 83-48 shows the analysis results when the detailed cross section is selected (Delta L = 0.00265 in and Stress-X = 178.57 psi).

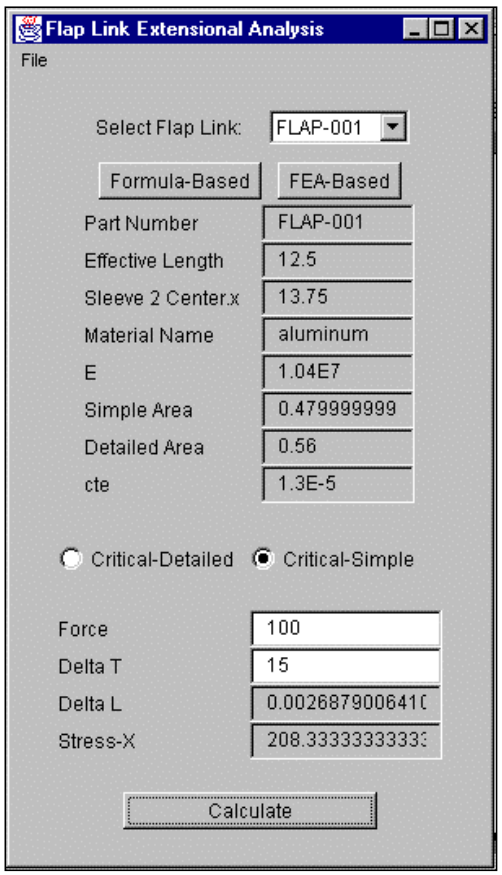


Figure 83-47: Flap Link Extensional Analysis Application (Showing Formula-Based Analysis Results when Simple Cross Section is Selected)

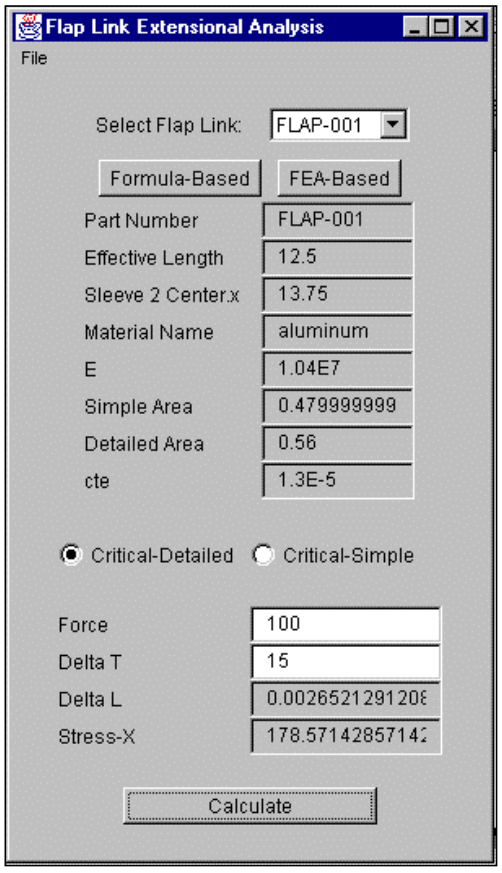


Figure 83-48: Flap Link Extensional Analysis Application (Showing Formula-Based Analysis Results when Detailed Cross Section is Selected)

When the finite-element analysis is selected (as shown in Figure 83-49), a preprocessing file is created and sent to the finite-element analysis program for solution (this particular application creates an Ansys Prep7 file, which is sent to Ansys for processing). Figure 83-50 shows a portion of a sample Prep7 file created with this application. Figure 83-51 is a screen shot of Ansys displaying the solved model (this screen shot is showing the solution for the axial deformation).

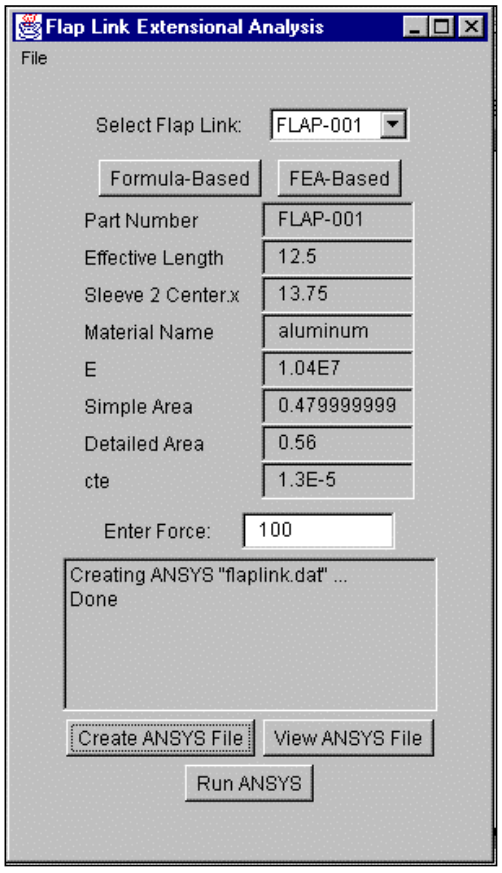


Figure 83-49: Flap Link Extensional Analysis Application (Showing Finite Element Analysis Selected)

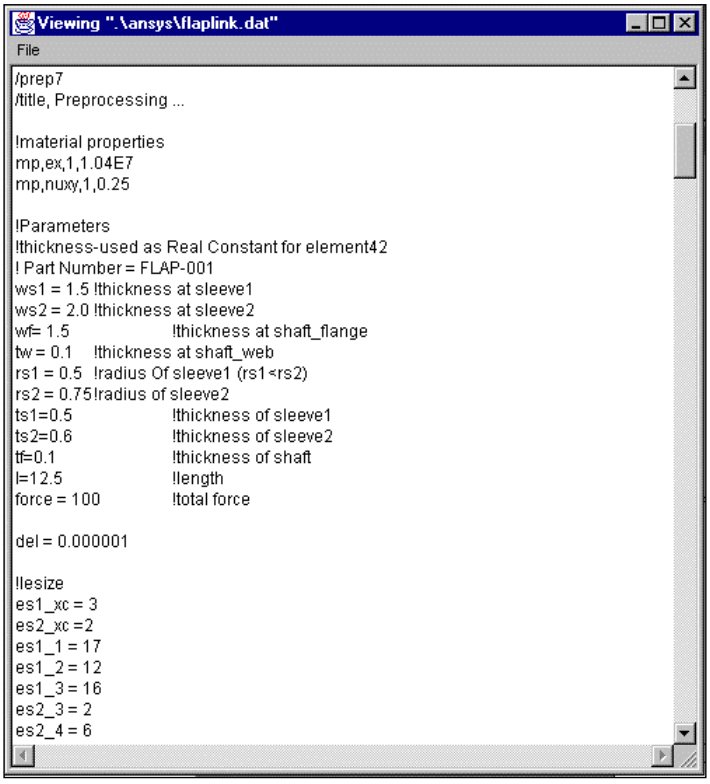


Figure 83-50: Preprocessing File (Prep7) Sent to Ansys

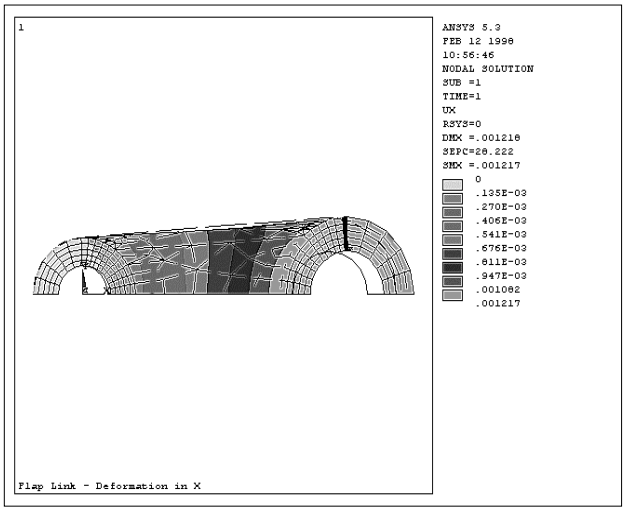


Figure 83-51: Flap Link Finite Element Analysis Results

As expected, the finite-element analysis requires more detailed information about the geometry of the flap link than the formula-based solution. The values queried in order to create the preprocessing file that is sent to the finite-element analysis program for processing are:

```
String partNumber = selectedFlapLink.
      getStringInstance( "part_number" ).
      getStringValue();

double E = selectedFlapLink.
      getObjectInstance( "material" ).
      getMultiLevelInstance( "stress_strain_model" ).
      getObjectInstance("temperature_independent_linear_elastic"
      ).
      getRealInstance( "youngs_modulus" ).
      getRealValue( );

double poissons = selectedFlapLink.
      getObjectInstance( "material" ).
      getMultiLevelInstance( "stress_strain_model" ).
      getObjectInstance("temperature_independent_linear_elastic"
      ).
      getRealInstance( "poissons_ratio" ).
      getRealValue( );

double L = selectedFlapLink.
      getRealInstance( "effective_length" ).
      getRealValue();

double ws1 = selectedFlapLink.
      getObjectInstance( "sleeve_1" ).
      getRealInstance( "width" ).
      getRealValue();
```

```
double ws2 = selectedFlapLink.
      getObjectInstance( "sleeve_2" ).
      getRealInstance( "width" ).
      getRealValue();

double rs1 = selectedFlapLink.
      getObjectInstance( "sleeve_1" ).
      getRealInstance( "radius" ).
      getRealValue();

double rs2 = selectedFlapLink.
      getObjectInstance( "sleeve_2" ).
      getRealInstance( "radius" ).
      getRealValue();

double ts1 = selectedFlapLink.
      getObjectInstance( "sleeve_1" ).
      getRealInstance( "thickness" ).
      getRealValue();

double ts2 = selectedFlapLink.
      getObjectInstance( "sleeve_2" ).
      getRealInstance( "thickness" ).
      getRealValue();

double tw = selectedFlapLink.
      getObjectInstance( "shaft" ).
      getMultiLevelInstance( "critical_cross_section" ).
      getObjectInstance( "simple" ).
      getRealInstance( "tw" ).
      getRealValue();

double tf = selectedFlapLink.
      getObjectInstance( "shaft" ).
```

```
        getMultiLevelInstance( "critical_cross_section" ).
        getObjectInstance( "simple" ).
        getRealInstance( "tf" ).
        getRealValue();

double wf = selectedFlapLink.
        getObjectInstance( "shaft" ).
        getMultiLevelInstance( "critical_cross_section" ).
        getObjectInstance( "simple" ).
        getRealInstance( "wf" ).
        getRealValue();
```

Figures 83-52 and 83-53 show two PBAMs (Subsection 9) illustrating how the Flap Link APM is used in this application to support these two types of analysis models (formula- and FEA-based, respectively). In these figures, analysis models are represented as boxes with "connection points" each corresponding to an analysis variable. An APM attribute being used by an analysis model is represented by a line connecting the APM attribute to one of the connection points of the analysis model.

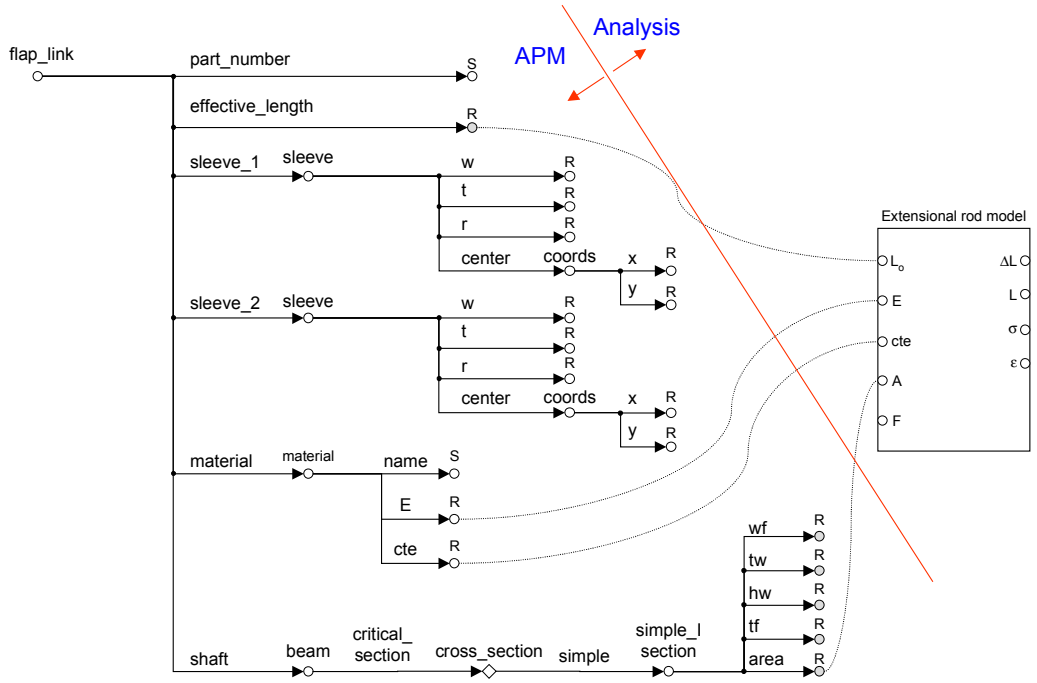


Figure 83-52: Flap Link APM Usage by Formula-Based Analysis Model (only partial APM shown)

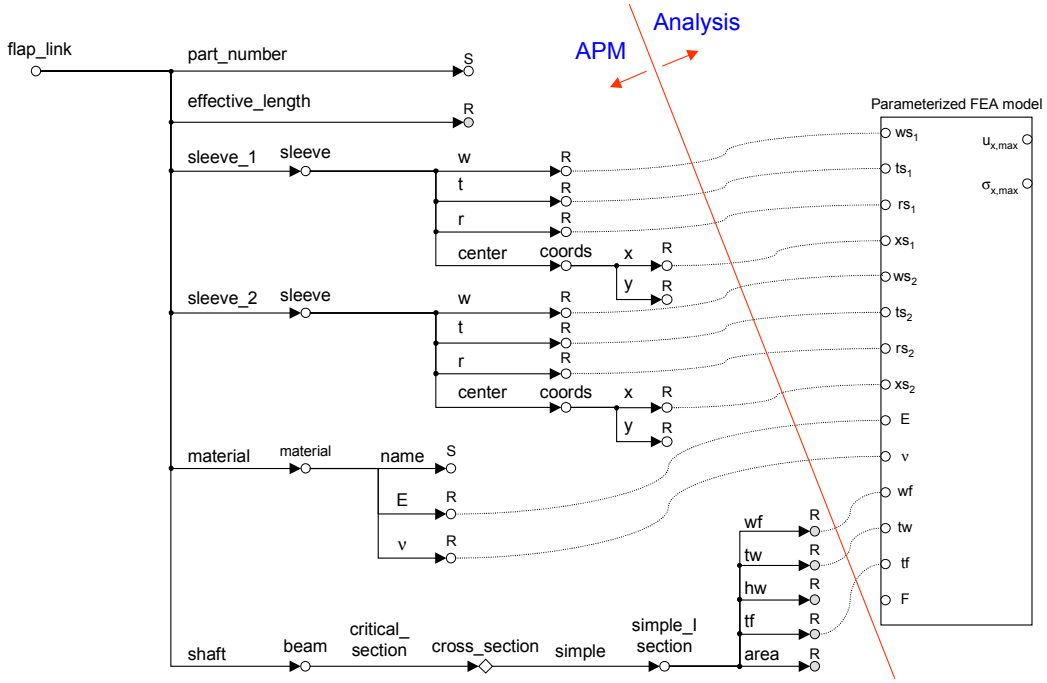


Figure 83-53: Flap Link APM Usage by FEA-Based Analysis Model (only partial APM shown)

Back Plate Analysis and Synthesis Application

The Back Plate Analysis and Synthesis Application was developed for the main purpose of demonstrating the capability of the APM Protocol operations to support two-way integration between design and analysis, in other words, to support *analysis* as well as *synthesis*. In order to achieve such integration, the APM provides - as it will be illustrated with the examples presented in this subsection - the capability of dynamically changing the input/output directions of the relations defined in the APM.

The Back Plate Analysis and Synthesis Application simulates the design and synthesis of a plate with two holes such as the one introduced in Subsection 86 and therefore this application is based on the APM defined in that section. The goal of this application is to help the analyst[58] determine the geometry of the plate such that the stress and elongation it undergoes when subjected to a given axial load are within specified values. For this purpose, the application allows the analyst to:

1. Fill in the values of the geometric attributes of the plate and run a formula-based tensional analysis to check the stress and elongation obtained with the proposed geometry (design checking), and/or
2. Set target values for the stress or elongation and determine the geometry of the plate required to produce those results (design synthesis)[59].

To illustrate this, consider the scenario represented in Figures 83-54 and 83-55. In this scenario, the analyst loads/sets some design values and checks them with some engineering analysis (steps 1 through 4, labeled "Design Checking 1"). Then he sets a desired analysis result and runs the analysis "in reverse" to obtain target design values (steps 5 and 6, labeled "Synthesis"). Finally, he refines one of the obtained design values and reruns the analysis in the original direction to verify that the design is okay (steps 7 and 6, labeled "Design Checking 2"). The following paragraphs walk through this process, discussing each step in more detail.

---

[58] In this discussion, the user of this application is referred to as *the analyst*, even though he or she is also performing design.

[59] Design checking and synthesis are discussed in Subsection 5.

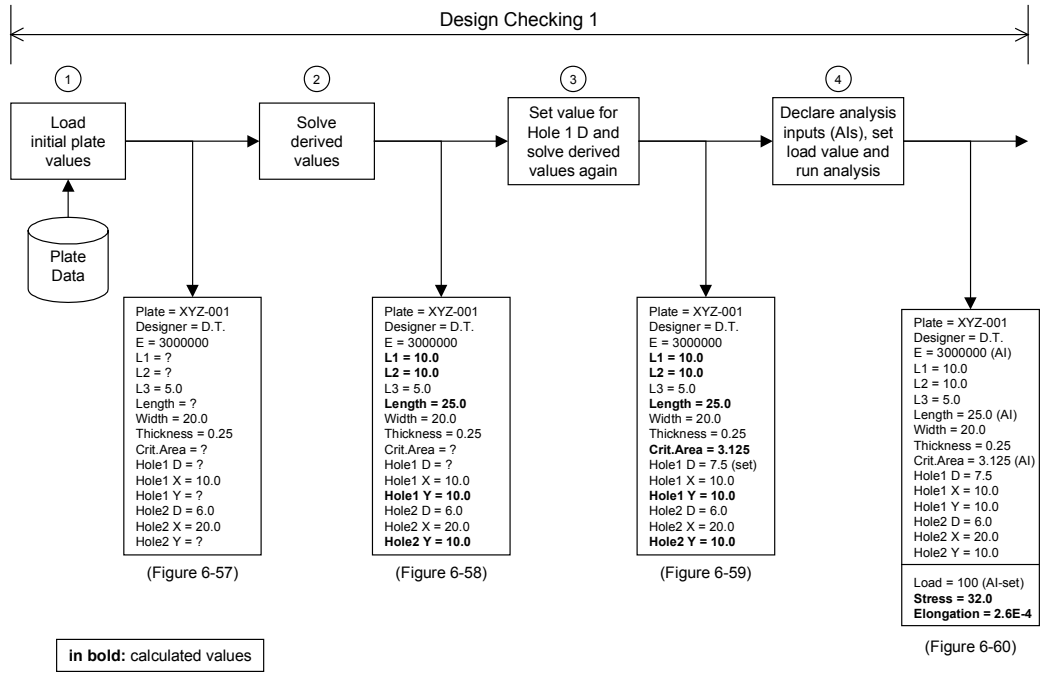


Figure 83-54: Plate Design/Synthesis Scenario

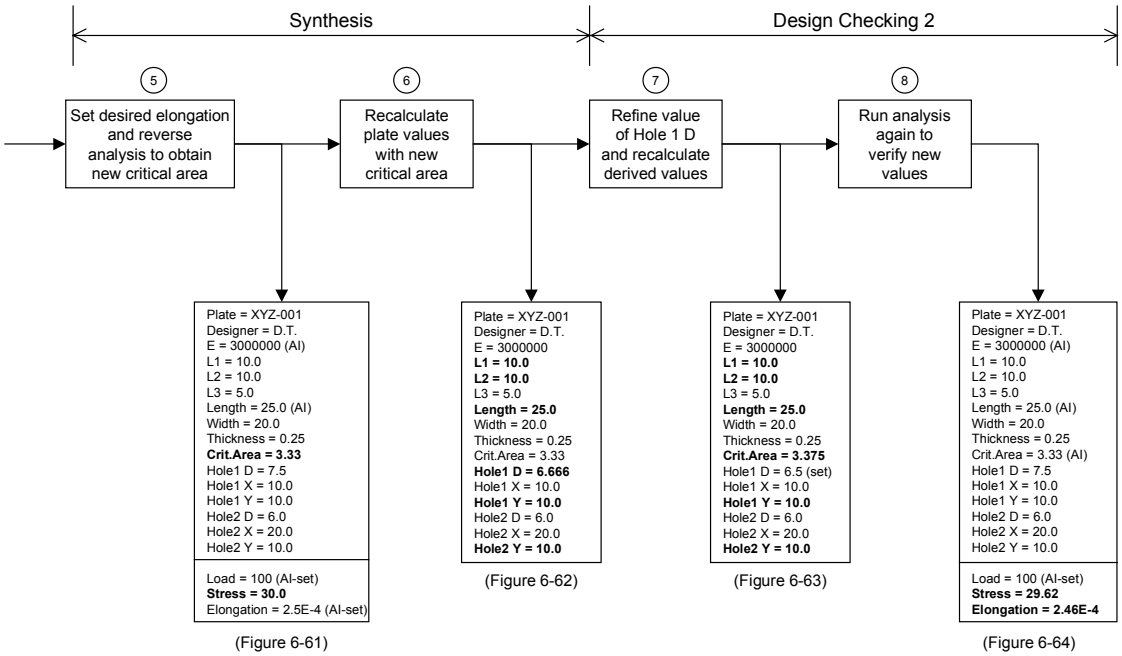


Figure 83-55: Plate Design/Synthesis Scenario (continued)

In step 1, the analyst loads some preliminary design data from existing source data files. The resulting screen is shown in Figure 83-56. The APM definition and data are loaded in the same way they were loaded in the previous applications. In this case, since the Back Plate APM has three source sets (`back_plate_geometric_model`, `back_plate_material_-data`, and `back_plate_employee_data`), the application will prompt the user for three source data files. Since it is not known in advance which instances have value in the source data files and which do not, the application must check if an instance has value before attempting to display it. For example, the code to query the initial value of attribute **`Length`** in order to display it in the first screen shown by the application (Figure 83-56) is the following:

```
APMRealInstance lengthInstance = plateInstance.
      getRealInstance( "length" );
if( lengthInstance.hasValue() )
{
      double length = lengthInstance.getRealValue();
      lengthField.setText( Double.toString( length ) );
}
```

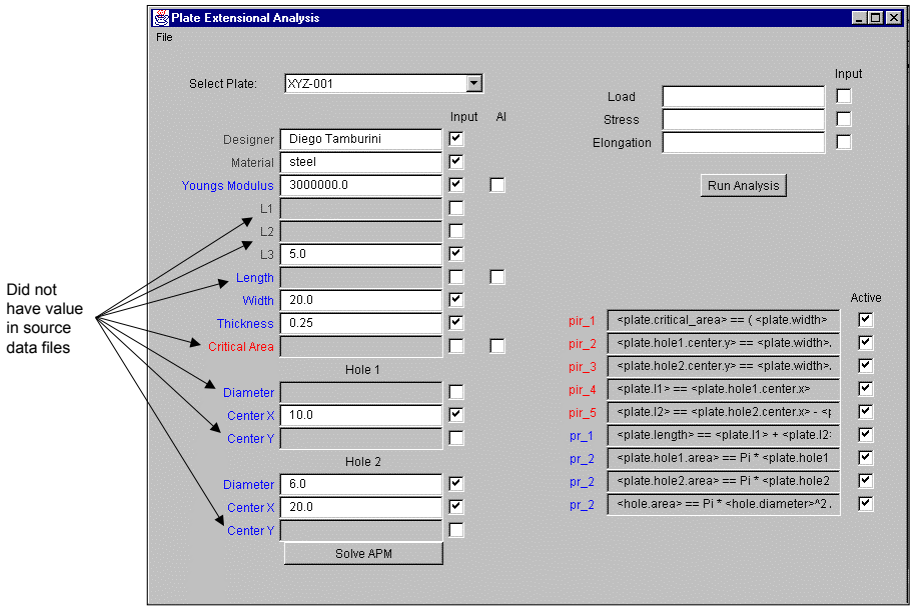


Figure 83-56: Back Plate Analysis and Synthesis Application (showing initial data screen)

In the code above, if attribute `length` of the instance **`plateInstance`** of domain `plate` has value, it will be queried with method **`getRealValue`** and displayed in the appropriate field (**`lengthField`**) using the standard Java method **`setText`**. Otherwise it will be left blank. In addition, if the attribute does have value, its corresponding "Input" checkbox will be checked (method **`loadSourceSetData`** initializes the value of attribute **`isInput`** of all the instances of **`APMPrimitiveDomainInstances`** that have value to **`true`** when the data is first read). In Java, this can be done with the standard method **`setState`**, as follows (considering attribute `length` as an example):

```
lengthIsInputCheckbox.setState( lengthInstance.isInput() );
```

Where **`lengthIsInputCheckbox`** is the "Input" checkbox next to the **`Length`** field and **`setState`** is a Java method that checks the box if the argument passed to it (in this case **`lengthInstance.isInput()`**) is **`true`**. In summary, if attribute `length` of the plate instance read from the source data file has value, the value will be displayed in the **`Length`** field, the attribute will be declared as an input by method **`loadSourceSetData,`** and its corresponding "Input" checkbox will be checked.

As also shown in Figure 83-56, after loading the initial set of data some of the geometric parameters of the plate (**`L1`**, **`L2`**, **`Length`**, **`Hole 1 Diameter`**, **`Hole 1 Center Y`**, and **`Hole 2 Center Y`**) do not have value yet. Thus, the goal of the analyst is to determine the values of these parameters in order to complete the design. However, these values cannot be assigned at will since, as it is almost invariably the case, the design must meet some specified performance criteria. For example, in this case, the specification requires that the stress be lower than 32 psi and the elongation lower than 2.5E-4 in when an axial load of 100 lb is applied.

Next (step 2), the analyst checks if any of the missing values can be determined at this point by using the current values and the relations defined in the APM. So he or she clicks the "Solve APM" button (as shown in Figure 83-57), initiating a series of constraint-solving requests that will attempt to determine these missing values. For example, the following code will try to solve for the value of **`Length`** if it is an output (in this example, **`Length`** is initially an output because its value was not originally populated in the source data file):

```
if( lengthInstance.isOutput() )
{
```

```
        APMRealInstance lengthInstance = plateInstance.
              getRealInstance( "length" );
        setFieldValue( lengthInstance , lengthField );
    }
```

Function **setFieldValue** used in the code above is a convenience function defined just for this application (in other words, it is *not* and APM Protocol operation). It takes an **APMRealInstance** and a **TextField** (a Java graphical component used to display text) as arguments. The definition of this function is:

```
    private void setFieldValue( APMRealInstance instance ,
                                TextField textField )
    {
        int numberOfSolutions;
        if( instance.hasValue() )
        {
            double value = instance.getRealValue();
            textField.setText( Double.toString( value ) );
        }
        else
        {
            numberOfSolutions = instance.trySolveForValue();
            if( numberOfSolutions > 0 )
            {
                double value = instance.getRealValue();
                textField.setText( Double.toString( value ) );
            }
            else
                textField.setText( "" );
        }
    }
```

If the instance passed as an argument to this function has value, the value is displayed in the **`TextField`**.[60] Otherwise, method **`trySolveForValue`** will attempt to solve for it using the relations defined in the APM. If a solution is found (that is, if **`numberOfSolutions > 0`**) it is displayed in the **`TextField`**, otherwise an empty string is displayed.

Figure 83-57 also shows the results obtained when the analyst clicks the "Solve APM" button. Some of the attributes that did not have value in the initial screen (**`L1`**, **`L2`**, **`Length`**, **`Hole 1 Center Y`**, and **`Hole 2 Center Y`**) now do. This is because the constraint-solving operation was able to find the necessary relations and input values to calculate these values. For example, **`L1`** was found using relation `pir_4` of the Back Plate APM (see Subsection 86) defined as follows:

```
pir_4 : "<l1> == <hole1.center.x>";
```

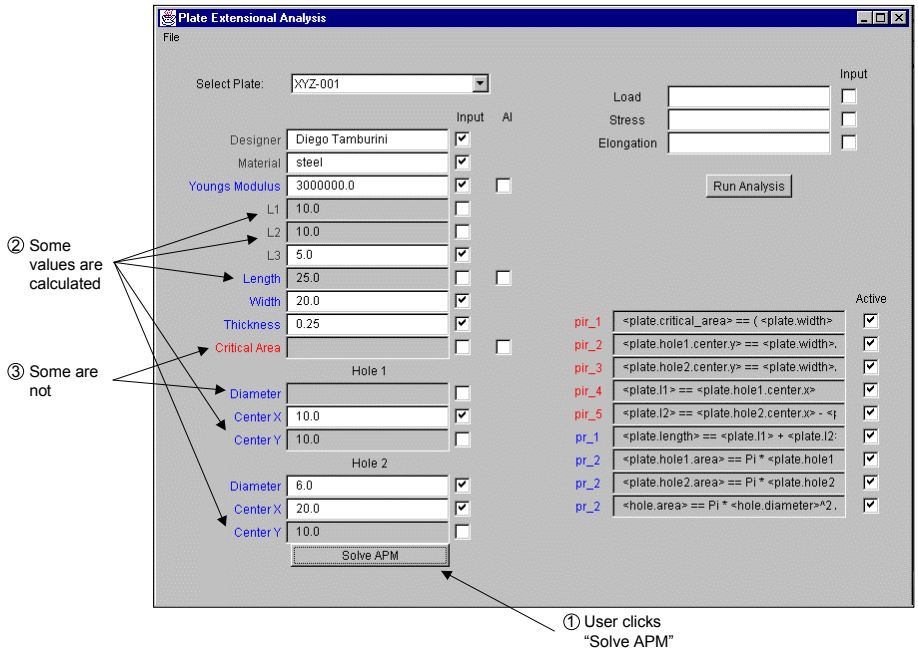


Figure 83-57: Back Plate Analysis and Synthesis Application (preliminary results after clicking "Solve APM")

[60] Not only input instances have value; an output instance could also have value.

In the relation above, since **Hole 1 Center X** had a value of **10** in then the resulting value for **L1** was also equal to **10** in (Figure 83-57).

However, the constraint-solving operation could not determine the values of **Critical Area** and **Hole 1 Diameter**. The only relation in the APM in which the critical area is involved is relation `pir_1`, defined as follows:

```
pir_1 : "<critical_area> == ( <width> - <hole1.diameter> ) *
<thickness>";
```

in which **Width** and **Thickness** have value (**20** in and **0.25** in, respectively). However, neither **Critical Area** nor **Hole 1 Diameter** can be determined because they are both unknowns participating in the same relation.

Therefore, the analyst proceeds to determine the missing values. For this purpose, in step 3 he or she sets a value of **7.5** in for **Hole 1 Diameter** (as shown in Figure 83-58) and clicks the "Solve APM" button again. Now, with this additional value, the value of **Critical Area** can be determined with relation `pir_1` above. As shown in the figure, the resulting value of **Critical Area** when **Hole 1 Diameter** is set to **7.5** in is **3.125** in^2.

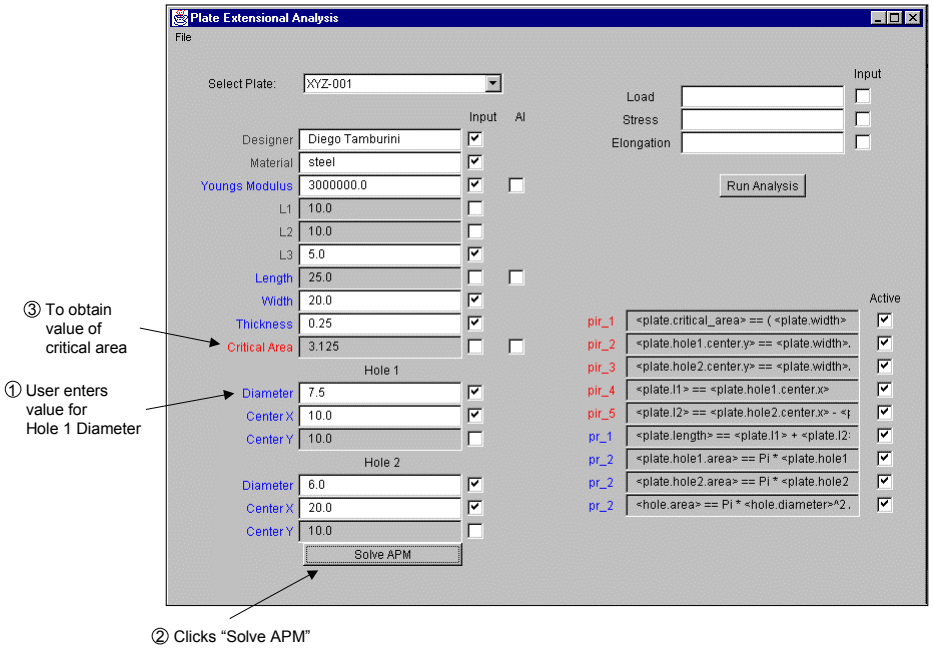


Figure 83-58: Back Plate Analysis and Synthesis Application (showing value for hole 1 diameter entered by the analyst)

At this point, the values of all the attributes of the plate have been determined. However, this is only a preliminary design that still needs to be checked against a series of engineering analyses (in this example, only a tension analysis is performed). Thus, in step 4, the analyst proceeds to declare which attributes are analysis inputs and which are analysis outputs. In this example, he or she selects **`Young's Modulus`**, **`Length`**, **`Critical Area`** and **`Load`** as analysis inputs (by clicking in the "Analysis Input" checkbox as shown in Figure 83-59), and **`Stress`** and **`Elongation`** as analysis outputs. Next, he or she enters a value of **`100`** lb for **`Load`** and clicks "Run Analysis", initiating the analysis[61]. The resulting **`Stress`** (as shown in the figure) is **`32.0`** psi and the resulting **`Elongation`** is **`2.666E-4`** in. The stress is below the specified value of 35 psi, but the elongation is still too high (it is above the specified maximum value of 2.5E-4 in). Therefore, the preliminary design has to be revised.

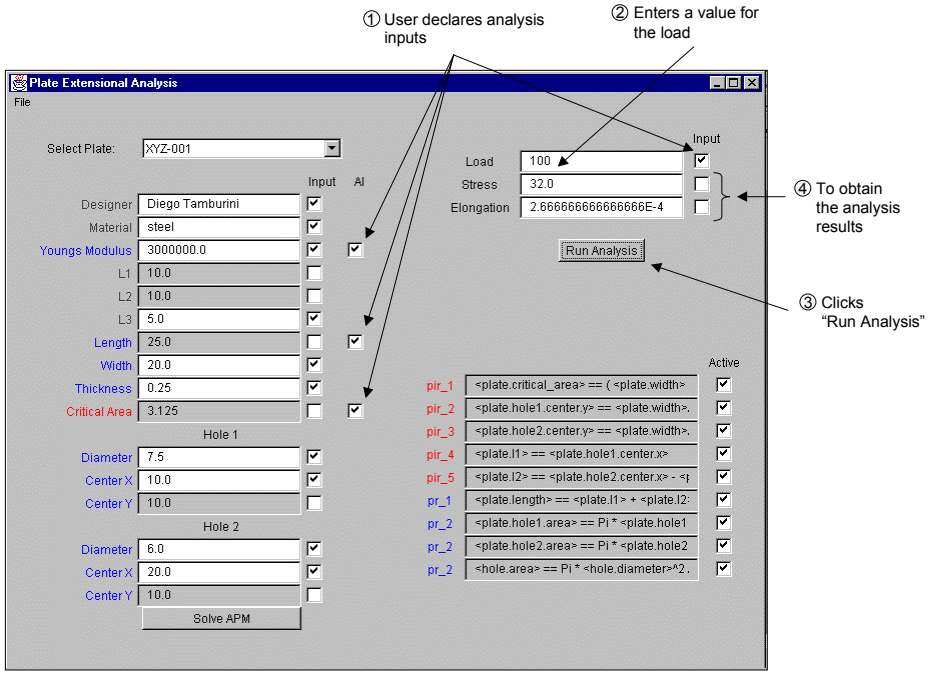


Figure 83-59: Back Plate Analysis and Synthesis Application (showing selected analysis inputs)

[61] Note: this application contains two connected but distinct constraint systems. The first deals only with APM attributes, while the second deals with APM attributes and the attributes of the analysis model. A checkbox in the "Input" column is checked when the corresponding attribute is an input in the APM's constraint system. On the other hand, a checkbox in the "Analysis Input" column is checked when the attribute is an input in the analysis model. Some attributes, such as **`Young's Modulus`**, **`Length`** and **`Critical Area`** participate in both constraint systems (hence the reason why they have two checkboxes). These attributes are the ones that connect both constraint systems, and they will normally act as inputs in one system and outputs in the other.

At this point the analyst has two possible courses of action to correct the design:

1. Iteratively adjust the design values of the plate (**Hole 1 Diameter**, **Width** or **Thickness**) until a **Critical Area** that produces acceptable values of **Stress** and **Elongation** is obtained, or
2. Specify *target* values for **Stress** and **Elongation** and enter them in the analysis model to obtain the corresponding target value for **Critical Area**. Then use the APM relations "in reverse" to obtain the values of the design parameters (**Hole 1 Diameter**, **Width** or **Thickness**) required to obtain this target critical area.

The first approach, as discussed in Chapter 1, is known as *iterative synthesis*. In this approach, the APM relations are used in their "natural direction".[62] The natural direction of an APM relation (or set of APM relations) is arbitrarily defined to be "from" the design attributes "to" the idealized attributes. In other words, when design attributes are used as inputs to obtain idealized attributes. The second approach is known as *synthesis*. In this approach, the APM relations are used in their "inverse direction" or "in reverse", meaning that idealized attributes are used as inputs to obtain design attributes.

A combination of these two approaches used to obtain the final design of the plate will be illustrated in the screen shots that follow. In step 5 (shown in Figure 83-60), the analyst selects **Elongation**, **Load**, **Young's Modulus** and **Length** as the analysis inputs in order to obtain **Critical Area**. He or she sets the value of **Elongation** to the specified target value of **2.5E-4** in and runs the analysis to determine the value of the **Critical Area** (**3.3333** in^2) that causes such deformation. Next, in step 6 (shown in Figure 83-61) the analyst selects **Critical Area** as an input and clicks the "Solve APM" button to get a value of **6.6666** in for **Hole 1 Diameter** (a smaller diameter than before, as expected)[63]. Since a hole with a diameter of 6.666 in may be difficult to manufacture, in step 7 the analyst changes the value to **6.5** in (as shown in Figure 83-62), declares it as an input and **Critical**

[62] Keep in mind that, in this example, there are two constraint systems: the APM constraint system and the analysis constraint system. The idea of the "natural direction" as explained here refers to the APM constraint system only. However, the concept could be extended to the analysis system by defining the "natural direction" to be when the inputs of the analysis are APM attributes (idealized or not) and environmental parameters (such as temperatures and applied loads), and the outputs are performance or behavioral parameters (such as stresses and deformations).

[63] This time, the APM relations were used "in reverse", since **Critical Area** (an idealized attribute) was used as an input to obtain **Hole 1 Diameter** (a design attribute).

Area as an output, and clicks the "Solve APM" again[64]. The critical area changes from **3.333** in^2 to **3.375** in^2**.** Finally, in step 8 the analyst performs a final analysis run to make sure that the stress and elongation are within the specified limits. For this, he or she sets **Critical Area** as an analysis input (by clicking the "Analysis Input" checkbox next to the **Critical Area** field, as shown in Figure 83-63), unchecks the **Elongation** checkbox (to make it an analysis output again), and clicks "Run Analysis". The results obtained are **29.62** psi for **Stress** and **2.46E-4** in for **Elongation**, both below the maximum allowable values specified of 32 psi and 2.5E-4 in, respectively. Since the values for stress and elongation are below the specified limits, the analyst considers the design to be satisfactory.

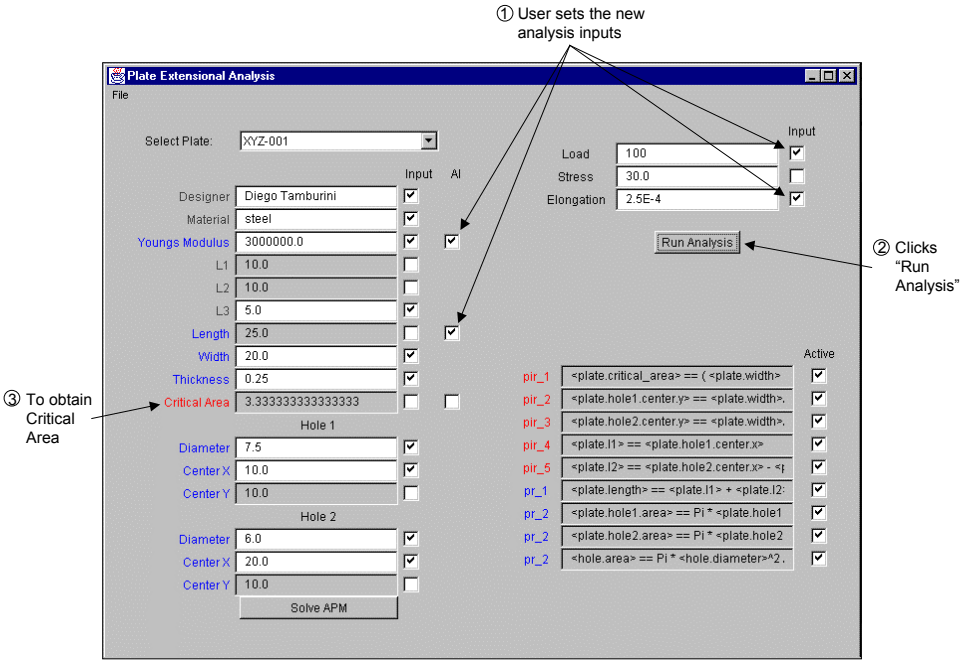


Figure 83-60: Back Plate Analysis and Synthesis Application (showing the analysis being performed to obtain critical area)

[64] This time, the APM relations were used again in their "natural direction".

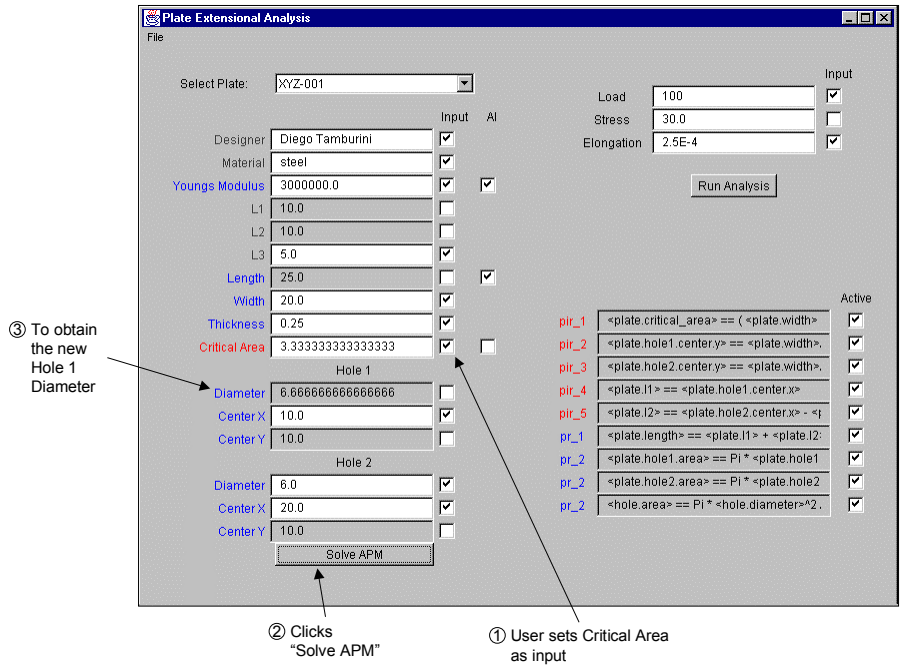


Figure 83-61: Back Plate Analysis and Synthesis Application (showing critical area selected as input)

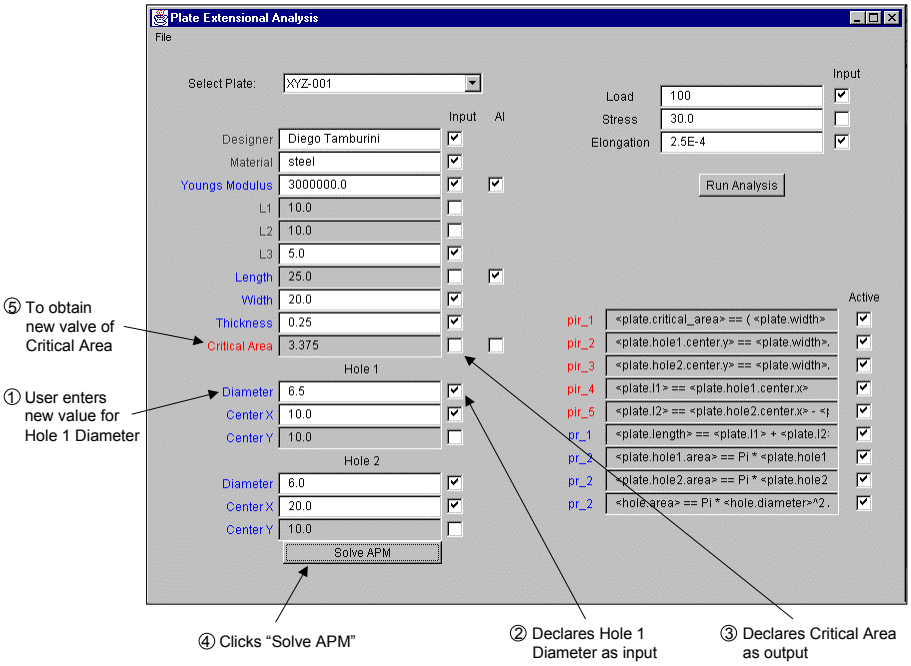


Figure 83-62: Back Plate Analysis and Synthesis Application (showing new value for diameter of hole 1)

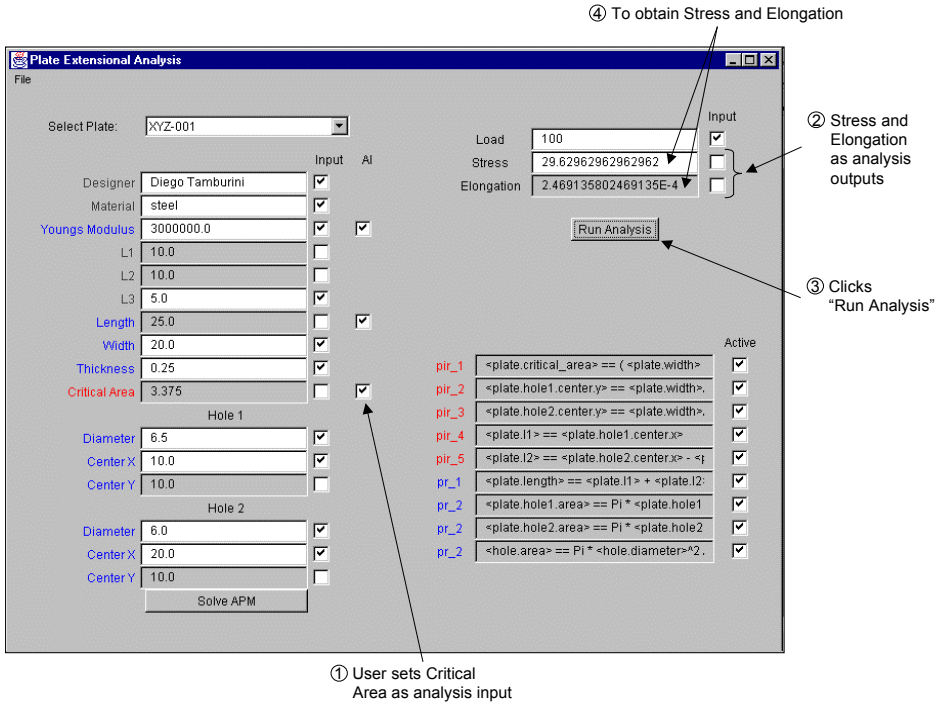


Figure 83-63: Back Plate Analysis and Synthesis Application (showing final analysis being performed)

When the "Input" checkbox for an attribute is checked (to change it from input to output or from output to input), the constraint network must be updated to reflect this change. As discussed in Subsection 81 methods **`APMRealInstance.setAsOutput`** and **`APMRealInstance.setAsInput`** take care of this update process. For example, when the **`Critical Area`** "Input" checkbox is checked, the following code is performed:

```
if( criticalAreaIsInputCheckbox.getState() == true )
      criticalAreaInstance.setAsInput();
else
      criticalAreaInstance.setAsOutput();
```

Here, when the Java function **`getState`** returns **`true`** (meaning that the "Input" checkbox of **`Critical Area`** was just checked to change the attribute from output to input), **`criticalAreaInstance`** is changed to input with method **`setAsInput`**. Conversely, if **`getState`** returns **`false`** (meaning that the "Input" checkbox of **`Critical Area`** was just unchecked to change the attribute from input to output) **`criticalAreaInstance`** is set as

output with method **setAsOuput**. Recall from Subsection 81 that these methods take into account the effect that changing an instance from input to output (or vice versa) has on the rest of the instances in the constraint network.

Recall also from the discussion of Subsection 81 that the behavior specified by the APM Protocol for methods **APMRealInstance.setAsInput** and **APMRealInstance.setAs-Output** is rather naïve, since the task of determining which input/output combinations are valid for a given constraint network is left entirely to the programmer or to the analyst. As a consequence, it is possible to specify invalid input/output combinations that lead to conflicting solutions or to no solutions at all. For example, there is not a mechanism in place to prevent the analyst from setting both **Critical Area** and **Hole 1 Diameter** as inputs, and then entering inconsistent values for them that violate relation `pir_1`.

This leads to the next important feature also demonstrated by this application, which is the ability to *relax* relations. Relaxing a relation essentially means removing it from the constraint network, so that the relation is not taken into account when the constraint-solving request is built (see Subsection 81). This is useful when the analyst wants to ignore a constraint imposed by a relation without having to modify the APM itself (for example, because the change applies to only one situation, or he or she is just performing what-if analysis). To illustrate this relaxation process suppose that, for some reason, the analyst wants to set the **Hole 2 Center Y** to **9.5** in instead of the current value of **10** in. However, relation `pir_3`, defined as follows:

```
pir_3 : "<hole2.center.y> == <width>/2";
```

constrains the y coordinate of the center of hole 2 to be half the width of the plate. Thus, he or she must relax this relation if he or she wants to be able to set an arbitrary value for **Hole 2 Center Y** without causing a conflict. In order to relax the relation, he or she clicks the "Active" checkbox next to relation `pir_3` (as shown in Figure 83-64), deactivating the relation and temporarily removing it from the constraint network. When this checkbox is unchecked, method **ConstraintNetworkRelation.setActive** is used to relax the relation as follows:

```
relation3.setActive( false );
```

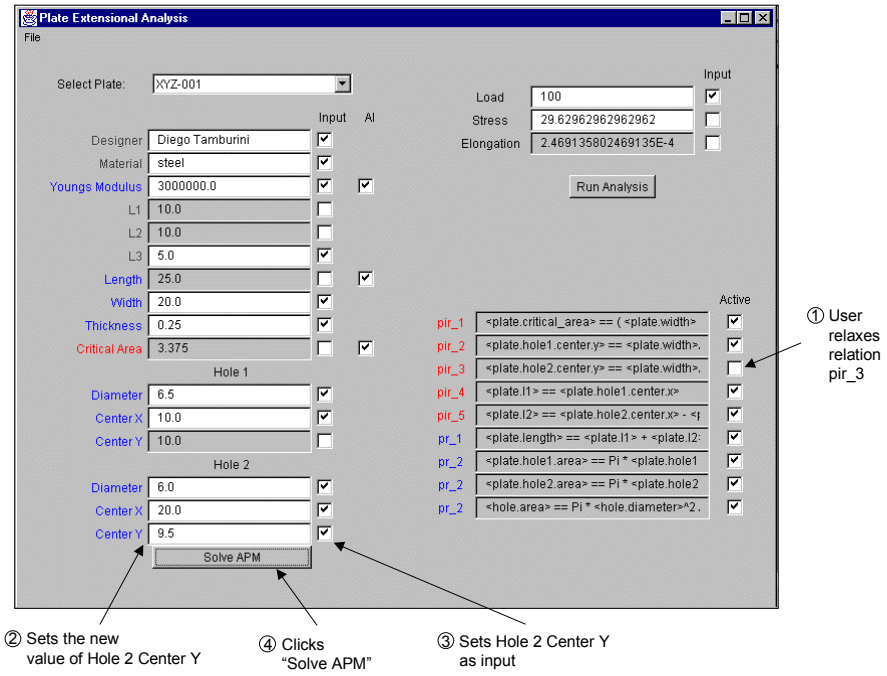


Figure 83-64: Back Plate Analysis and Synthesis Application (showing relaxation of a relation)

Now, the analyst can enter an arbitrary value for **`Hole 2 Center Y`** without causing a conflict. He or she clicks "Solve APM" one last time and, as expected, nothing happens (if the relation had still been active, **`Width`** would have changed its value, causing an inconsistency with the value of **`Hole 1 Center Y`** in relation `pir_2`).

APM Browser

The APM Browser was developed in this work with the purpose of providing an example of a generic APM client application[65]. Generic APM client applications, as discussed at the beginning of Section 89, are not tied to a specific APM and therefore work with any valid APM definition. Thus, the APM Browser is designed to load and display the structure (source sets, domains, attributes, relations, and source set links) of any APM - as well as instances of the domains defined in this APM – despite the fact that the structure of the APM is not known to the APM Browser until run time.

[65] A more recent version of the APM Browser presented here – called COB Tree Browser – is presented in Section 113.

This tool can be used, for example, to aid in the creation and debugging of new APM definitions, even before APM-Specific client applications have been developed for them. In fact, the APM Browser was used extensively in the creation and debugging of the APM definitions presented in Section 84.

The APM Browser can also display the values of the attributes defined in the APM; both those populated directly from the source data files and those calculated at run time by the constraint solver using the relations defined in the APM. Therefore, the APM Browser can also be used to check the validity of the APM relations and the results they produce.

The Flap Link APM presented in Subsection 85 will be used to illustrate the various tasks that can be performed with the APM Browser. The first action must be to load the APM definition. This step is the same load APM definition step of all the applications presented in this section, and therefore uses the same method **`APMInterface.loadAPMDefinitions`**. Figure 83-65 is a screen shot of the APM Browser while the APM definitions are being loaded. In this figure, the application is prompting the user for the file that contains the APM definition.

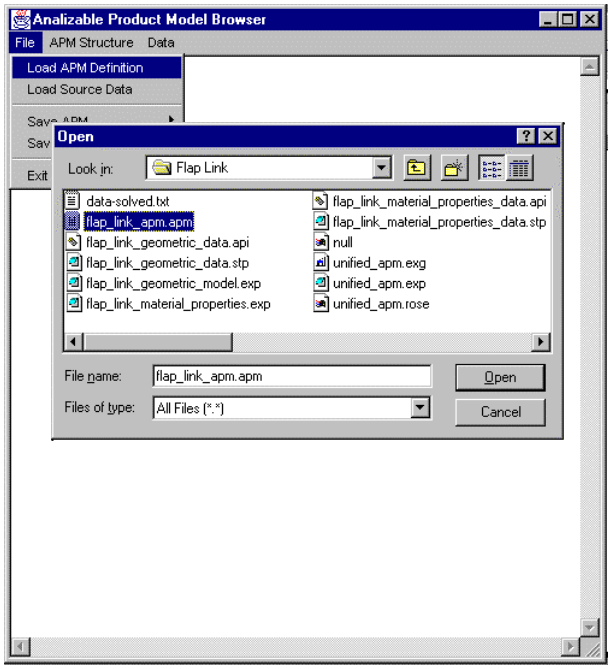


Figure 83-65: APM Browser (loading the APM definitions)

Once the APM definitions have been loaded, the user has the choice of displaying the structure of the APM in two different ways: before or after the source set links are applied. In the "unlinked" display the domains are listed grouped by source sets. In the "linked" display the domains are listed in a single, unified source set that results from combining the various source sets of the APM using the source set link definitions. This linked version of the APM may be useful, for example, to review the resulting structure of the APM and data types of its attributes after the source set links are applied. The unlinked APM structure display is created with method **`APMInterface.printUnlinkedAPMDefinitions`** and the linked display is created with method **`APMInterface.printLinkedAPMDefinitions`**. Figures 83-66 and 83-67 show the unlinked and the linked APM structures, respectively, being displayed for the Flap Link APM. Notice, for example, that attribute `material` in domain `flap_link` is of type `STRING` in the fist screen (before linking) and of type `material` in the second screen (after linking) (recall from Subsection 85 that the two source sets of this APM are linked through this attribute).

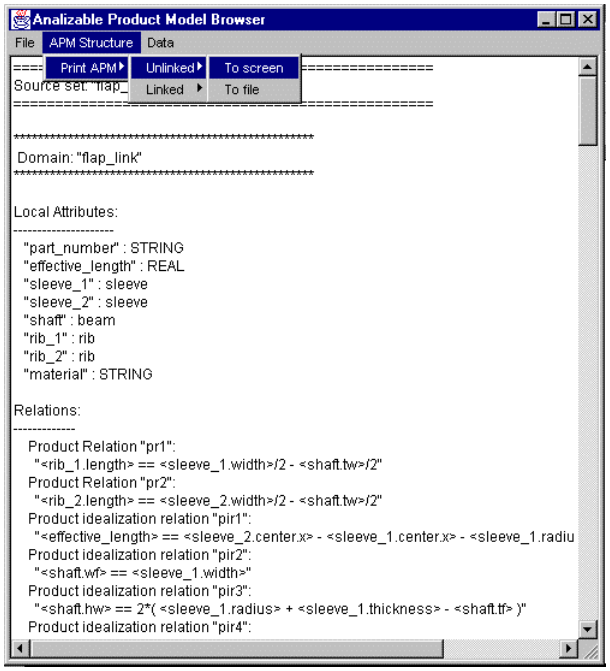


Figure 83-66: APM Browser (listing unlinked APM definitions)

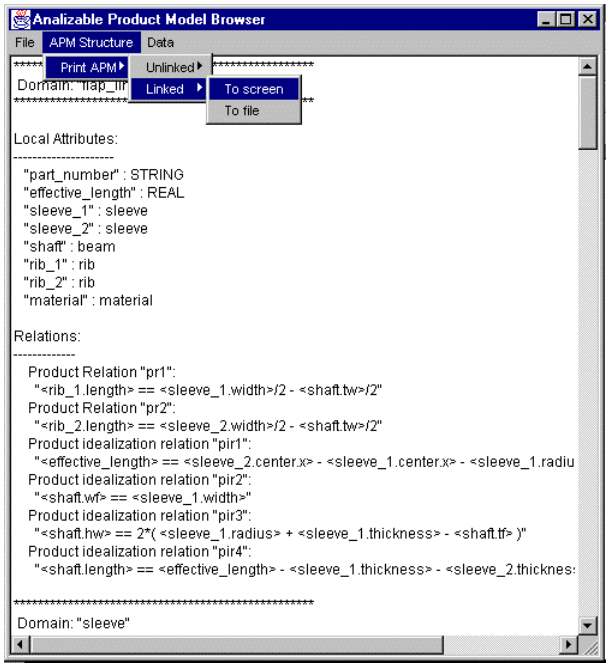


Figure 83-67: APM Browser (listing linked APM definitions)

At this point, as shown in Figure 83-68, the linked version of the APM definition may also be saved to a file in APM-S format. Method **`APMInterface.saveLinkedAPMDefinition`** is used for this purpose. Figure 83-69 shows a portion of the linked Flap Link APM created by the APM Browser with this method. Notice in this figure that there is only one source set (`unified_apm`) and that attribute `material` of domain `flap_link` is of type `material`.

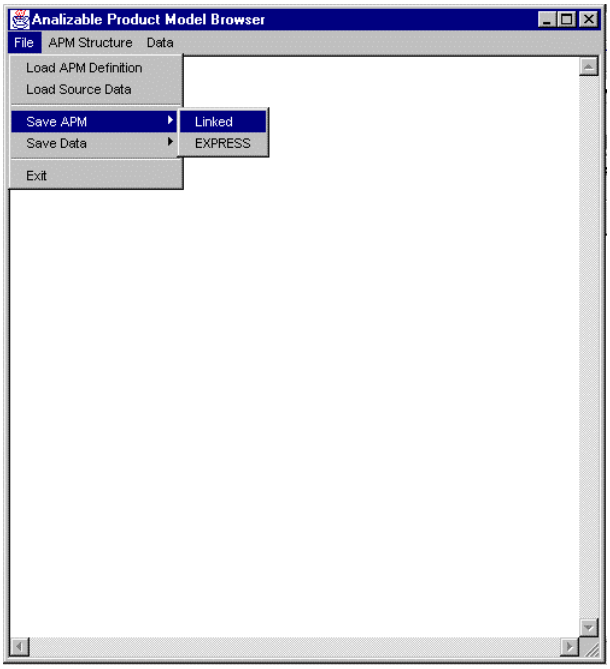


Figure 83-68: APM Browser (creating linked APM definition)

```
APM flap_link;

SOURCE_SET unified_apm ROOT_DOMAIN flap_link;

DOMAIN flap_link;
  IDEALIZED part_number : STRING;
  IDEALIZED effective_length : REAL;
  sleeve_1 : sleeve;
  sleeve_2 : sleeve;
  shaft : beam;
  rib_1 : rib;
  rib_2 : rib;
  material : material;

 PRODUCT_IDEALIZATION_RELATIONS
  pir1 : "<effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> - <sleeve_1.radius> - <sleeve_2.radius>";
  pir2 : "<shaft.wf> == <sleeve_1.width>";
  pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> - <shaft.tf> )";
  pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness> - <sleeve_2.thickness>";

 PRODUCT_RELATIONS
  pr1 : "<rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2";
  pr2 : "<rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2";

END_DOMAIN;


DOMAIN material;
  IDEALIZED name : STRING;
  stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

(* Other domains omitted *)

END_SOURCE_SET;

END_APM;
```

Figure 83-69: Linked APM Definition Created by the APM Browser (partial)

Recall from Subsection 55 that APM-S definitions can be translated into lexical EXPRESS. This may be useful, for example, if the source data is in STEP P21 format, since the compiled EXPRESS schemas are needed by the APM Browser in order to be able to read the data. In addition, EXPRESS-G diagrams can be automatically generated from the lexical EXPRESS using commercial STEP utilities. These diagrams may be useful to communicate the structure of domain-specific models (recall the discussion about generic and domain-specific models in Section 41). The APM Browser may be used to obtain the lexical EXPRESS corresponding to the APM-S definition of the APM. Method **`APMInterface.exportToExpress`** is used for this purpose. The result is one EXPRESS schema for each source set defined in the APM, plus an additional schema corresponding to the "unified" (linked) version. For example, the unified EXPRESS-G diagrams for the Flap Link APM shown in Figures 83-9 and 83-10 were obtained this way. For the Flap Link APM three EXPRESS schemas are created: schemas `flap_link_geometric_model` (created from the first source set), `flap_link_material_properties` (created from the second source set), and `unified_apm` (created from the linked version of the APM). Figure 83-70 shows a portion of the EXPRESS version of the linked APM created by the APM Browser.

```
SCHEMA unified_apm;

ENTITY flap_link;
   (* ESSENTIAL *) part_number : STRING;
   (* IDEALIZED *) effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   material : material;
(* WHERE
 (* PRODUCT IDEALIZATION RELATIONS *)
   pir1 : <effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> - <sleeve_1.radius> - <sleeve_2.radius>;
   pir2 : <shaft.wf> == <sleeve_1.width>;
   pir3 : <shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> - <shaft.tf> );
   pir4 : <shaft.length> == <effective_length> - <sleeve_1.thickness> - <sleeve_2.thickness>;

 (* PRODUCT RELATIONS *)
   pr1 : <rib_1.length> == <sleeve_1.width>/2 - <shaft.tw>/2;
   pr2 : <rib_2.length> == <sleeve_2.width>/2 - <shaft.tw>/2;

*)
END_ENTITY;


ENTITY material;
   (* ESSENTIAL *) name : STRING;
   stress_strain_model : material_levels;
END_ENTITY;


(* Other entities omitted *)

END_SCHEMA;
```

Figure 83-70: EXPRESS Version of the Linked APM Created by the APM Browser (partial)

The next step is to load the source set data corresponding to this APM. As in the other applications presented in this section, method **`APMInterface.loadSourceSetData`** is used for this purpose. As shown in Figure 83-71, the application will prompt the user for two source set data files (one for each source set defined in the Flap Link APM). The data in these files can be either in APM-I or STEP P21 format.

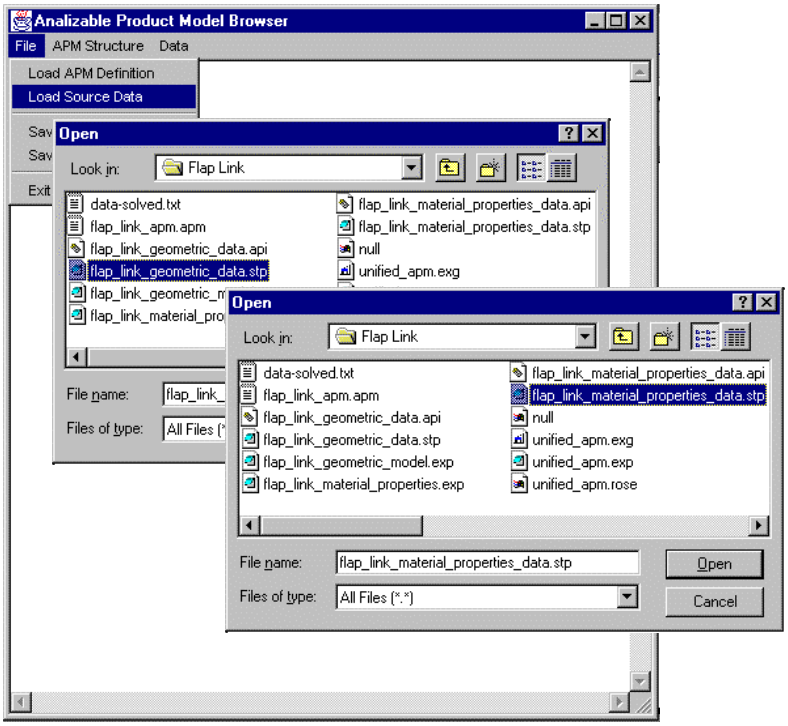
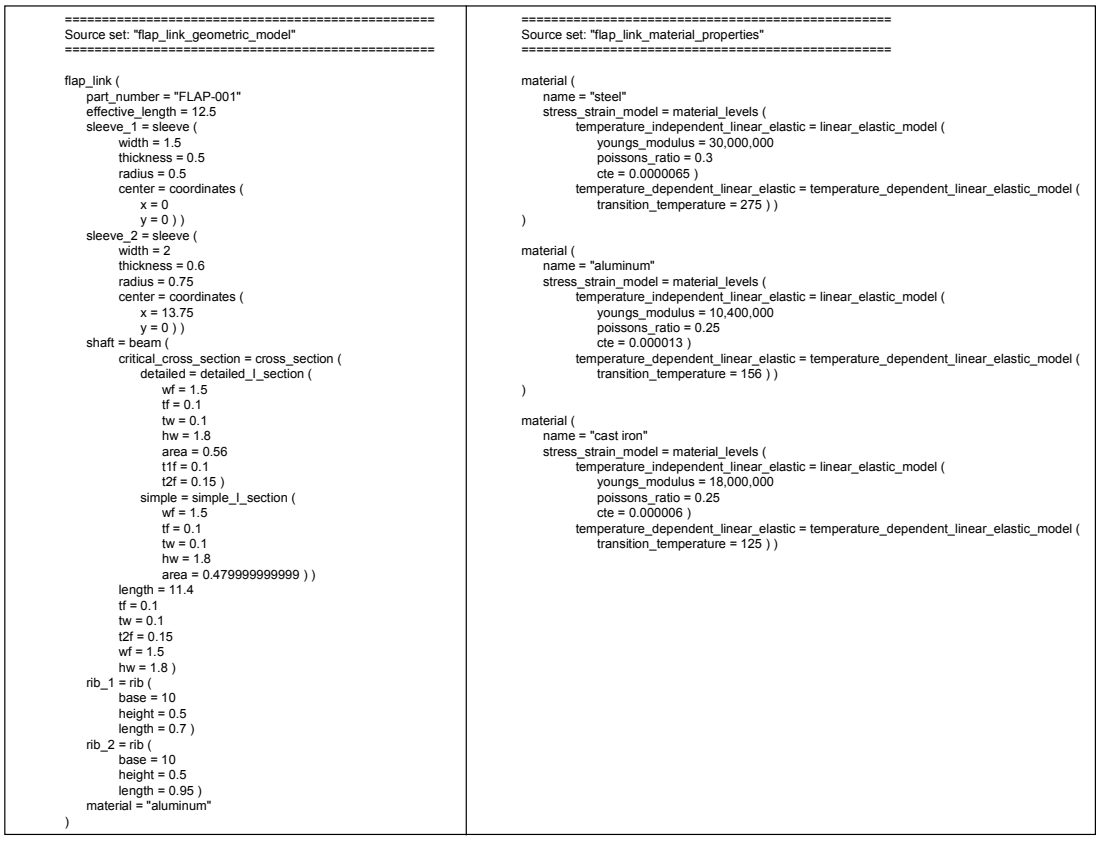


Figure 83-71: APM Browser (loading source set data)

Once the data is loaded, the user has the choice to display it before or after linking. The unlinked display is generated by method **`APMInterface.printUnlinkedAPMInstances`** and the linked display by method **`APMInterface.printLinkedAPMInstances`**. Since these two methods use method **`APMRealInstance.getRealValue`** (see Subsection 81) to display the values of the real attributes, they will trigger a constraint-solving attempt for each attribute that does not have value in the source data files. Figure 83-72 shows the APM Browser being used to display the unlinked data and Figure 83-73 is a closer look at the data displayed. Notice that all the attributes displayed have value. Notice also that attribute `material` of the instance of domain `flap_link` shown ("`FLAP-001`") has a string value of

"`aluminum`" (indicating that the data has not been linked yet). This string will be used to link this instance of `flap_link` with the second instance of `material` of source set `flap_link_material_properties` being shown in the figure.

Figure 83-72: APM Browser (displaying unlinked data)



```
==================================================
Source set: "flap_link_geometric_model"
==================================================

flap_link (
    part_number = "FLAP-001"
    effective_length = 12.5
    sleeve_1 = sleeve (
        width = 1.5
        thickness = 0.5
        radius = 0.5
        center = coordinates (
            x = 0
            y = 0 ) )
    sleeve_2 = sleeve (
        width = 2
        thickness = 0.6
        radius = 0.75
        center = coordinates (
            x = 13.75
            y = 0 ) )
    shaft = beam (
        critical_cross_section = cross_section (
            detailed = detailed_I_section (
                wf = 1.5
                tf = 0.1
                tw = 0.1
                hw = 1.8
                area = 0.56
                t1f = 0.1
                t2f = 0.15 )
            simple = simple_I_section (
                wf = 1.5
                tf = 0.1
                tw = 0.1
                hw = 1.8
                area = 0.479999999999 ) )
        length = 11.4
        tf = 0.1
        tw = 0.1
        t2f = 0.15
        wf = 1.5
        hw = 1.8 )
    rib_1 = rib (
        base = 10
        height = 0.5
        length = 0.7 )
    rib_2 = rib (
        base = 10
        height = 0.5
        length = 0.95 )
    material = "aluminum"
)
```

```
==================================================
Source set: "flap_link_material_properties"
==================================================

material (
    name = "steel"
    stress_strain_model = material_levels (
        temperature_independent_linear_elastic = linear_elastic_model (
            youngs_modulus = 30,000,000
            poissons_ratio = 0.3
            cte = 0.0000065 )
        temperature_dependent_linear_elastic = temperature_dependent_linear_elastic_model (
            transition_temperature = 275 ) )
)

material (
    name = "aluminum"
    stress_strain_model = material_levels (
        temperature_independent_linear_elastic = linear_elastic_model (
            youngs_modulus = 10,400,000
            poissons_ratio = 0.25
            cte = 0.000013 )
        temperature_dependent_linear_elastic = temperature_dependent_linear_elastic_model (
            transition_temperature = 156 ) )
)

material (
    name = "cast iron"
    stress_strain_model = material_levels (
        temperature_independent_linear_elastic = linear_elastic_model (
            youngs_modulus = 18,000,000
            poissons_ratio = 0.25
            cte = 0.000006 )
        temperature_dependent_linear_elastic = temperature_dependent_linear_elastic_model (
            transition_temperature = 125 ) )
```

Figure 83-73: Unlinked Data Created by APM Browser (some instances omitted)

Figure 83-74 shows the APM Browser being used to display the linked data and Figure 83-75 is a closer look at the data displayed. Notice that, as a result of linking, attribute `material` now points to an instance of domain `material` whose `name` is "`aluminum`" (instead of directly pointing to the string "`aluminum`" as before linking).

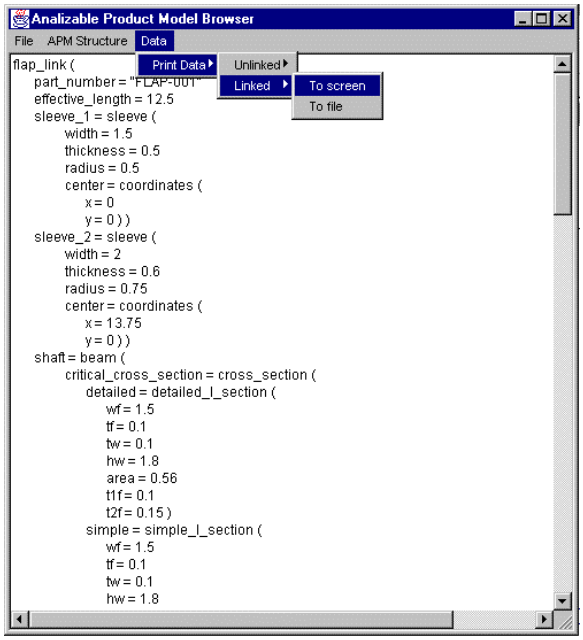


Figure 83-74: APM Browser (displaying linked data)

```
flap_link (
    part_number = "FLAP-001"
    effective_length = 12.5
    sleeve_1 = sleeve (
        width = 1.5
        thickness = 0.5
        radius = 0.5
        center = coordinates (
            x = 0
            y = 0 ) )
    sleeve_2 = sleeve (
        width = 2
        thickness = 0.6
        radius = 0.75
        center = coordinates (
            x = 13.75
            y = 0 ) )
    shaft = beam (
        critical_cross_section = cross_section (
            detailed = detailed_I_section (
                wf = 1.5
                tf = 0.1
                tw = 0.1
                hw = 1.8
                area = 0.56
                t1f = 0.1
                t2f = 0.15 )
            simple = simple_I_section (
                wf = 1.5
                tf = 0.1
                tw = 0.1
                hw = 1.8
                area = 0.479999999999 ) )
        length = 11.4
        tf = 0.1
        tw = 0.1
        t2f = 0.15
        wf = 1.5
        hw = 1.8 )
```

```
    rib_1 = rib (
            base = 10
            height = 0.5
            length = 0.7 )
        rib_2 = rib (
            base = 10
            height = 0.5
            length = 0.95 )
        material = material (
            name = "aluminum"
            stress_strain_model = material_levels (
                temperature_independent_linear_elastic = linear_elastic_model (
                    youngs_modulus = 10,400,000
                    poissons_ratio = 0.25
                    cte = 0.000013 )
                temperature_dependent_linear_elastic = temperature_dependent_linear_elastic_model (
                    transition_temperature = 156 ) ) )
)
```

Figure 83-75: Linked Data Created by APM Browser (only “FLAP-001” instance shown)

Since a significant computational effort may have been required to display the values of all the attributes (several constraint-solving operations may have been needed to do so), the user may want to save the results so that they can be loaded later without having to solve for the missing values again. For this purpose, the APM Browser provides the capability to save the solved data in APM-I format, as shown in Figure 83-76[66]. As discussed in Subsection 0, the data can be saved by source set (as it was originally loaded) or linked. Method **`APMInterface.saveInstancesBySourceSet`** is used to save the data in its unlinked form and method **`APMInterface.saveLinkedInstances`** is used to save the data in its linked form. Figure 83-77 shows the APM-I definition of "`FLAP-001`" before being loaded into the APM Browser (in its "original" or "unsolved" version) and after being displayed and saved with the APM Browser (in its "solved" version). Notice that all the attributes in the solved version have values, even those that had a question mark ("`?`") in the original version. The solved version can be saved and loaded later in another session, and the values will be displayed immediately without having to run the constraint solver again.

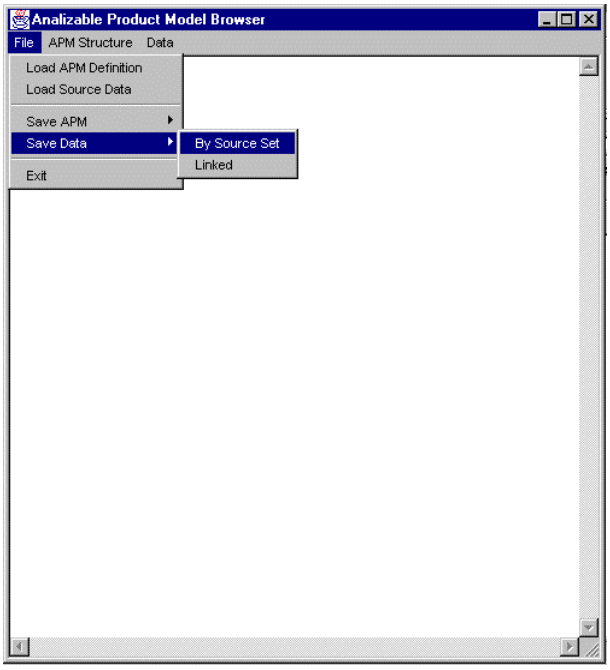


Figure 83-76: APM Browser (saving APM data by source set)

[66] Alternatively, the APM Browser could be extended to save data in STEP P21 as well.

Unsolved

```
INSTANCE_OF flap_link;
  part_number : "FLAP-001";
  effective_length : 12.5;
  sleeve_1.width : 1.5;
  sleeve_1.thickness : 0.5;
  sleeve_1.radius : 0.5;
  sleeve_1.center.x : 0.0;
  sleeve_1.center.y : 0.0;
  sleeve_2.width : 2.0;
  sleeve_2.thickness : 0.6;
  sleeve_2.radius : 0.75;
  sleeve_2.center.x : ?;
  sleeve_2.center.y : 0.0;
  shaft.length : ?;
  shaft.tf : 0.1;
  shaft.tw : 0.1;
  shaft.t2f : 0.15;
  shaft.wf : ?;
  shaft.hw : ?;
  shaft.critical_cross_section.detailed.wf : ?;
  shaft.critical_cross_section.detailed.tf : ?;
  shaft.critical_cross_section.detailed.tw : ?;
  shaft.critical_cross_section.detailed.hw : ?;
  shaft.critical_cross_section.detailed.area : ?;
  shaft.critical_cross_section.detailed.t1f : ?;
  shaft.critical_cross_section.detailed.t2f : ?;
  shaft.critical_cross_section.simple.wf : ?;
  shaft.critical_cross_section.simple.tf : ?;
  shaft.critical_cross_section.simple.tw : ?;
  shaft.critical_cross_section.simple.hw : ?;
  shaft.critical_cross_section.simple.area : ?;
  rib_1.base : 10.00;
  rib_1.height : 0.5;
  rib_1.length : ?;
  rib_2.base : 10.00;
  rib_2.height : 0.5;
  rib_2.length : ?;
  material : "aluminum";
END_INSTANCE;
```

Solved

```
INSTANCE_OF flap_link;
  part_number : "FLAP-001";
  effective_length : 12.5;
  sleeve_1.width : 1.5;
  sleeve_1.thickness : 0.5;
  sleeve_1.radius : 0.5;
  sleeve_1.center.x : 0.0;
  sleeve_1.center.y : 0.0;
  sleeve_2.width : 2.0;
  sleeve_2.thickness : 0.6;
  sleeve_2.radius : 0.75;
  sleeve_2.center.x : 13.75;
  sleeve_2.center.y : 0.0;
  shaft.critical_cross_section.detailed.wf : 1.5;
  shaft.critical_cross_section.detailed.tf : 0.1;
  shaft.critical_cross_section.detailed.tw : 0.1;
  shaft.critical_cross_section.detailed.hw : 1.8;
  shaft.critical_cross_section.detailed.area : 0.56;
  shaft.critical_cross_section.detailed.t1f : 0.1;
  shaft.critical_cross_section.detailed.t2f : 0.15;
  shaft.critical_cross_section.simple.wf : 1.5;
  shaft.critical_cross_section.simple.tf : 0.1;
  shaft.critical_cross_section.simple.tw : 0.1;
  shaft.critical_cross_section.simple.hw : 1.8;
  shaft.critical_cross_section.simple.area : 0.4799999999999999;
  shaft.length : 11.4;
  shaft.tf : 0.1;
  shaft.tw : 0.1;
  shaft.t2f : 0.15;
  shaft.wf : 1.5;
  shaft.hw : 1.8;
  rib_1.base : 10.0;
  rib_1.height : 0.5;
  rib_1.length : 0.7;
  rib_2.base : 10.0;
  rib_2.height : 0.5;
  rib_2.length : 0.95;
  material : "aluminum";
END_INSTANCE;
```

Figure 83-77: Flap Link Instance Before and After Solving

Finally, the APM Browser can also be used to save the data in its linked format. The resulting file is shown in Figure 83-78. Notice that the attributes of domain `material` from the second source set of the APM are now part of this instance (the last five attributes of the instance).

```
INSTANCE_OF flap_link;
  part_number : "FLAP-001";
  effective_length : 12.5;
  sleeve_1.width : 1.5;
  sleeve_1.thickness : 0.5;
  sleeve_1.radius : 0.5;
  sleeve_1.center.x : 0.0;
  sleeve_1.center.y : 0.0;
  sleeve_2.width : 2.0;
  sleeve_2.thickness : 0.6;
  sleeve_2.radius : 0.75;
  sleeve_2.center.x : 13.75;
  sleeve_2.center.y : 0.0;
  shaft.critical_cross_section.detailed.wf : 1.5;
  shaft.critical_cross_section.detailed.tf : 0.1;
  shaft.critical_cross_section.detailed.tw : 0.1;
  shaft.critical_cross_section.detailed.hw : 1.8;
  shaft.critical_cross_section.detailed.area : 0.56;
  shaft.critical_cross_section.detailed.t1f : 0.1;
  shaft.critical_cross_section.detailed.t2f : 0.15;
  shaft.critical_cross_section.simple.wf : 1.5;
  shaft.critical_cross_section.simple.tf : 0.1;
  shaft.critical_cross_section.simple.tw : 0.1;
  shaft.critical_cross_section.simple.hw : 1.8;
  shaft.critical_cross_section.simple.area : 0.4799999999999999;
  shaft.length : 11.4;
  shaft.tf : 0.1;
  shaft.tw : 0.1;
  shaft.t2f : 0.15;
  shaft.wf : 1.5;
  shaft.hw : 1.8;
  rib_1.base : 10.0;
  rib_1.height : 0.5;
  rib_1.length : 0.7;
  rib_2.base : 10.0;
  rib_2.height : 0.5;
  rib_2.length : 0.95;
  material.name : "aluminum";
  material.stress_strain_model.temperature_independent_linear_elastic.youngs_modulus : 1.04E7;
  material.stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
  material.stress_strain_model.temperature_independent_linear_elastic.cte : 1.3E-5;
  material.stress_strain_model.temperature_dependent_linear_elastic.transition_temperature : 156.0;
END_INSTANCE;
```

Figure 83-78: Linked APM Data (only "FLAP-001" shown)

APM-Design Tool Interfacing Tests

This section describes two preliminary tests performed by a team from the Engineering Information Systems Laboratory at the Georgia Institute of Technology (of which the author was a member) whose purpose was to demonstrate of interfacing the APM approach with commercial mechanical design tools (Chandrasekhar 1999). By the time of this writing, these tests were still under development and therefore the techniques described here were still being refined. Nevertheless, the state of these tests as captured in this thesis is complete enough to illustrate the main ideas.

These two tests illustrate the two possible approaches for interfacing an APM with a CAD tool discussed in Subsection 60. The first test demonstrates the *object-tagging* technique, whereas the second test demonstrates the *dimension-tagging* technique. In both cases, the subject is the same back plate presented in Subsection 86.

APM-Design Tool Interface Test Using the Object-Tagging Technique

In this test, the object-tagging technique described in Subsection 60 was used. Recall from this subsection that this technique consisted of manually tagging the objects within the solid modeler so that the semantic translator can recognize and map them into their corresponding APM instances. The names of the tags defined by the designer on the solid model must be consistent with the attribute names defined in the APM. Therefore, the designer must refer to the APM in order to obtain the names that he or she must use for the tagging process.

Figure 83-79 illustrates the sequence of steps performed in this test. In the first step (step 1), the designer creates the geometry of the plate using the CAD tool (Dassault Systemes' CATIA). Next (step 2), he or she proceeds to tag some of the geometric entities with APM-compatible names. The resulting tagged solid model is shown in Figure 83-80 (these tags are not just text annotations as shown in the figure; they are actually added to the object's data).

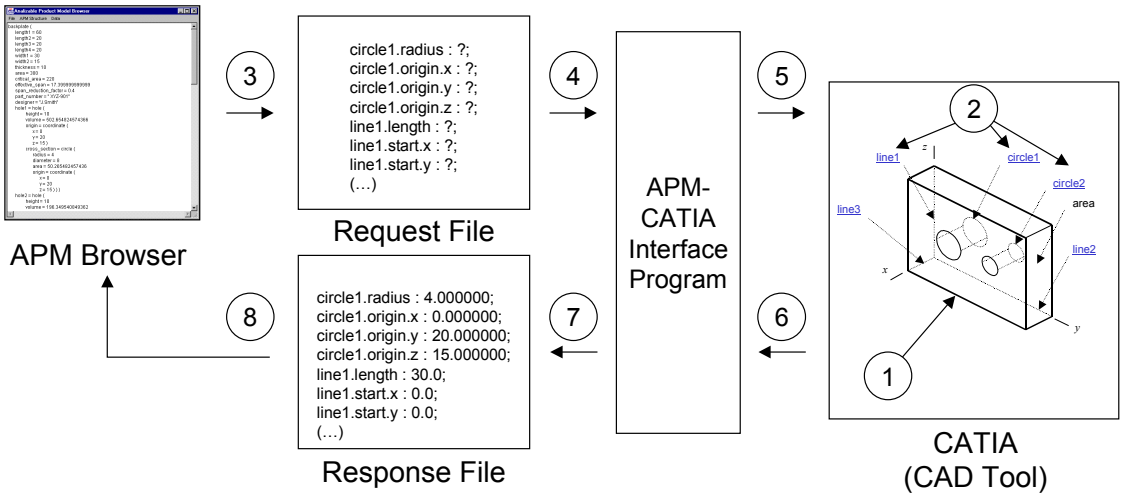


Figure 83-79: APM-Design Tool Interface Test Architecture

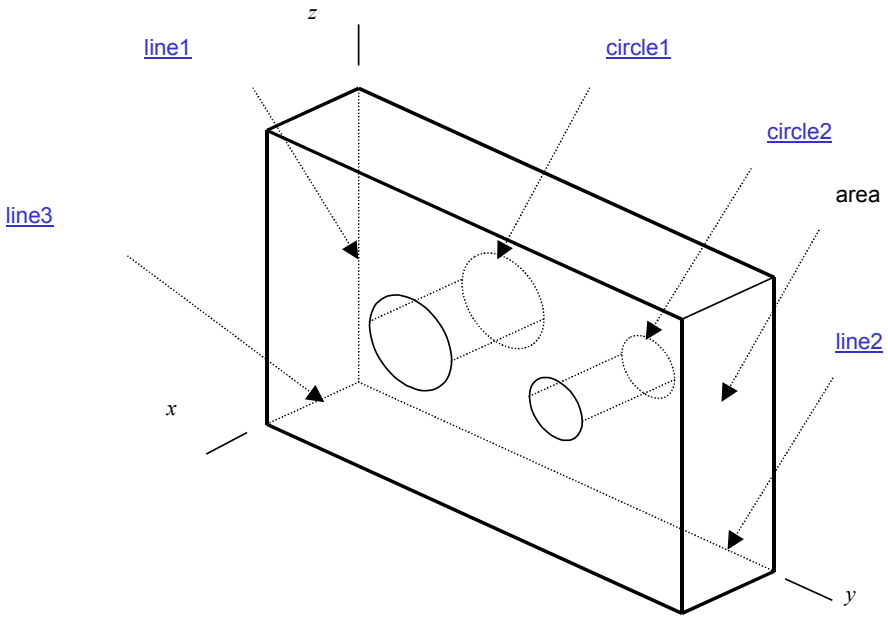


Figure 83-80: Object-Tagged Solid Model

The APM definition used for this test case, shown in Figures 83-81 and 83-82, is a modified version of the APM presented in Subsection 86. Although in this modified APM the design features of the plate are essentially the same, some additional domains (such as `line` and `circle`), attributes (such as `backplate.circle1` and `backplate.line1`) and relations (such as `prp1` through `prp11`) were added to facilitate the interfacing with the CAD tool. Figure 83-83 shows the back plate and its geometric attributes as defined in this APM.

```
APM backplate;

SOURCE_SET full_apm ROOT_DOMAIN backplate;

DOMAIN part;
  part_number : STRING;
  designer : STRING;
  origin : coordinate;
END_DOMAIN;

DOMAIN backplate SUBTYPE_OF part;
(* design features *)
  length1 : REAL;
  length2 :  REAL;
  length3 :  REAL;
  length4 :  REAL;
  width1 :  REAL;
  width2 :  REAL;
  thickness :  REAL;
  hole1 : hole;
  hole2 : hole;
  material : material;
  area : REAL;

(* geometric primitives *)
  circle1 : circle;
  circle2 : circle;
  line1 : line;
  line2 : line;
  line3 : line;

  IDEALIZED critical_area : REAL;
  IDEALIZED effective_span :  REAL;
  IDEALIZED span_reduction_factor : REAL;

 PRODUCT_RELATIONS
  pr1  : "<length1> == <length2> + <length3> + <length4>";
  pr2  : "<width2> == <width1> / 2";
  pr3  : "<hole1.height> == <thickness>";
  pr4  : "<hole1.origin.x> == <origin.x>";
  pr5  : "<hole1.origin.y> == <origin.y> + <length2>";
  pr6  : "<hole1.origin.z> == <origin.z> + <width2>";
  pr7  : "<hole2.height> == <thickness>";
  pr8  : "<hole2.origin.x> == <hole1.origin.x>";
  pr9  : "<hole2.origin.y> == <hole1.origin.y> + <length3>";
  pr10 : "<hole2.origin.z> == <hole1.origin.z>";
  pr11 : "<area> == <width1> * <thickness>";

(* relations with geometric primitives (in cad model) *)
  prp1  : "<length1> == <line2.length>";
  prp2  : "<width1> == <line1.length>";
  prp3  : "<thickness> == <line3.length>";
  prp4  : "<hole1.cross_section.radius> == <circle1.radius>";
  prp5  : "<hole1.origin.x> == <circle1.origin.x>";
  prp6  : "<hole1.origin.y> == <circle1.origin.y>";
  prp7  : "<hole1.origin.z> == <circle1.origin.z>";
  prp8  : "<hole2.cross_section.radius> == <circle2.radius>";
  prp9  : "<hole2.origin.x> == <circle2.origin.x>";
  prp10 : "<hole2.origin.y> == <circle2.origin.y>";
  prp11 : "<hole2.origin.z> == <circle2.origin.z>";

 PRODUCT_IDEALIZATION_RELATIONS
  pir1 : "<critical_area> == (<width1> - <hole1.cross_section.diameter>) * <thickness>";
  pir2 : "<effective_span> == <length3> - (<hole1.cross_section.radius> +
                              <hole2.cross_section.radius>) * <span_reduction_factor>";
END_DOMAIN;

(* --- part features --- *)

DOMAIN hole;
  origin : coordinate;
  cross_section :  circle;
  height :  REAL;
  volume : REAL;
 PRODUCT_RELATIONS
  r1 : "<volume> == <height> * <cross_section.area>";
  r2 : "<cross_section.origin.x> == <origin.x>";
  r3 : "<cross_section.origin.y> == <origin.y>";
  r4 : "<cross_section.origin.z> == <origin.z>";
END_DOMAIN;

DOMAIN material;
  (* actually a material model *)
  name : STRING;
  youngs_modulus : REAL;
  poissons_ratio : REAL;
  cte : REAL;
END_DOMAIN;
```

Figure 83-81: Modified Back Plate APM for the APM-CATIA Interface Test

```
(* --- geometric primitives --- *)

DOMAIN coordinate;
  x : REAL;
  y : REAL;
  z : REAL;
END_DOMAIN;

DOMAIN line;
  start : coordinate;
  end : coordinate;
  length : REAL;
 PRODUCT_RELATIONS
  r1: "<length> == ((<end.x> - <start.x>)^2 + (<end.y> - <start.y>)^2 +
                    (<end.z> - <start.z>)^2)^0.5";
END_DOMAIN;

DOMAIN circle;
  origin : coordinate;
  radius :  REAL;
  diameter :  REAL;
  area : REAL;
 PRODUCT_RELATIONS
  r1: "<diameter> == 2 * <radius>";
  r2: "<area> == 3.141516 * <radius>^2";
END_DOMAIN;

END_SOURCE_SET;

END_APM;
```

Figure 83-82: Modified Back Plate APM for the APM-CATIA Interface Test (continued)

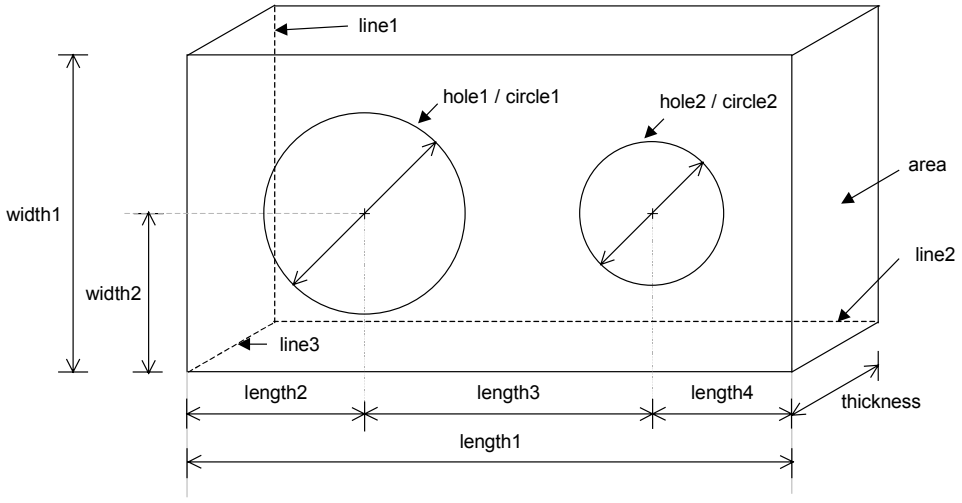


Figure 83-83: Back Plate Attributes

Admittedly, most of these additional attributes were added in order to overcome some limitations of the CATIA interface that was written for the test case. The main limitation

was that the only types of objects that the interface program could recognize - and therefore the only type of objects that could be tagged on the solid model - were circles and lines. In addition, only a limited set of attributes could be obtained from these objects. For circles, the only attributes that could be obtained were their radiuses and the coordinates of their centers. For lines, the only attributes that could be obtained were their lengths and the coordinates of their end points.

Hence, the APM had to be extended to include these two types of geometric primitives (circles and lines). For this purpose, domains `line` and `circle` were added (as shown in Figure 83-82). In addition, some attributes were also added to domain `backplate`, namely, attributes `circle1` and `circle2` (of type `circle`), and attributes `line1`, `line2` and `line3` (of type `line`). These attributes were added to be able to define relations to link the geometric primitives being read from the CAD model (`circles` and `lines`) with the design features of the back plate (`length1`, `length2`, etc.). For example, relation `prp1`, defined as follows:

```
prp1 : "<length1> == <line2.length>";
```

states that the `length` of the `line` tagged as "**`line2`**" in the CAD model (`line2.length`) corresponds to `length1` (a design feature) of `backplate`. In this case, the interface program will receive a request to query the value of attribute `length` of the `line` tagged as "**`line2`**". Then the APM client application will use relation `prp1` above to assign the value returned by this query to `length1` of `backplate`.

Continuing with the process illustrated in Figure 83-79, in step 3 the user uses the APM Browser to generate a "request" of the attributes that the APM needs from the solid model. This request is basically an APM-I file (labeled "Request File" in Figure 83-79 and shown in detail in Figure 83-84) containing all the names of the attributes defined in the APM. Notice that all the values of this request file (with the exception of `part_number`, `designer`, `span_reduction_factor` and `material.name`) are unknown.

```
DATA;

INSTANCE_OF backplate;

length1 : ? ;
length2 : ? ;
length3 : ? ;
length4 : ? ;
width1 : ? ;
width2 : ? ;
thickness : ? ;
area : ? ;
critical_area : ? ;
effective_span : ? ;
span_reduction_factor : 0.4 ;
part_number : " XYZ-901" ;
designer : "J.Smith" ;
hole1.height : ? ;
hole1.volume : ? ;
hole1.origin.x : ? ;
hole1.origin.y : ? ;
hole1.origin.z : ? ;
hole1.cross_section.radius : ? ;
hole1.cross_section.diameter : ? ;
hole1.cross_section.area : ? ;
hole1.cross_section.origin.x : ? ;
hole1.cross_section.origin.y : ? ;
hole1.cross_section.origin.z : ? ;
hole2.height : ? ;
hole2.volume : ? ;
hole2.origin.x : ? ;
hole2.origin.y : ? ;
hole2.origin.z : ? ;
hole2.cross_section.radius : ? ;
hole2.cross_section.diameter : ? ;
hole2.cross_section.area : ? ;
hole2.cross_section.origin.x : ? ;
hole2.cross_section.origin.y : ? ;
hole2.cross_section.origin.z : ? ;
material.name : "aluminium" ;
material.youngs_modulus : ? ;
material.poissons_ratio : ? ;
material.cte : ? ;
circle1.radius : ? ;
circle1.diameter : ? ;
circle1.area : ? ;
circle1.origin.x : ? ;
circle1.origin.y : ? ;
circle1.origin.z : ? ;
circle2.radius : ? ;
circle2.diameter : ? ;
circle2.area : ? ;
circle2.origin.x : ? ;
circle2.origin.y : ? ;
circle2.origin.z : ? ;
line1.length : ? ;
line1.start.x : ? ;
line1.start.y : ? ;
line1.start.z : ? ;
line1.end.x : ? ;
line1.end.y : ? ;
line1.end.z : ? ;
line2.length : ? ;
line2.start.x : ? ;
line2.start.y : ? ;
line2.start.z : ? ;
line2.end.x : ? ;
line2.end.y : ? ;
line2.end.z : ? ;
line3.length : ? ;
line3.start.x : ? ;
line3.start.y : ? ;
line3.start.z : ? ;
line3.end.x : ? ;
line3.end.y : ? ;
line3.end.z : ? ;
origin.x : ? ;
origin.y : ? ;
origin.z : ? ;

END_INSTANCE;

END_DATA;
```

Figure 83-84: Request File Sent to the Interface Program

Next (step 4), the request file is input into the CATIA-APM interface program, which interprets it and creates the corresponding API calls to get the requested values from the solid model (step 5). The interface program gets these values back from the solid model (step 6) and with them creates a "response" APM-I file (step 7). An example of this response file (labeled "Response File" in Figure 83-79) is shown in Figure 83-85. Notice that, in this file, many of the values that were requested in the request file (but not all) are now populated with values.

```
DATA;

INSTANCE_OF backplate;

length1 : ? ;
length2 : ? ;
length3 : ? ;
length4 : ? ;
width1 : ? ;
width2 : ? ;
thickness : ? ;
area : ? ;
critical_area : ? ;
effective_span : ? ;
span_reduction_factor : 0.4 ;
part_number : " XYZ-901" ;
designer : "J.Smith" ;
hole1.height : ? ;
hole1.volume : ? ;
hole1.origin.x : ? ;
hole1.origin.y : ? ;
hole1.origin.z : ? ;
hole1.cross_section.radius : ? ;
hole1.cross_section.diameter : ? ;
hole1.cross_section.area : ? ;
hole1.cross_section.origin.x : ? ;
hole1.cross_section.origin.y : ? ;
hole1.cross_section.origin.z : ? ;
hole2.height : ? ;
hole2.volume : ? ;
hole2.origin.x : ? ;
hole2.origin.y : ? ;
hole2.origin.z : ? ;
hole2.cross_section.radius : ? ;
hole2.cross_section.diameter : ? ;
hole2.cross_section.area : ? ;
hole2.cross_section.origin.x : ? ;
hole2.cross_section.origin.y : ? ;
hole2.cross_section.origin.z : ? ;
material.name : "aluminium" ;
material.youngs_modulus : 0.4 ;
material.poissons_ratio : 0.4 ;
material.cte : 0.4 ;
```

```
circle1.radius : 4.000000;
circle1.diameter : 8.0;
circle1.area : 50.2654;
circle1.origin.x : 0.000000;
circle1.origin.y : 20.000000;
circle1.origin.z : 15.000000;
circle2.radius : 2.500000;
circle2.diameter : 5.0;
circle2.area : 19.6349;
circle2.origin.x : 0.000000;
circle2.origin.y : 40.000000;
circle2.origin.z : 15.000000;
line1.length : 30.0;
line1.start.x : 0.0;
line1.start.y : 0.0;
line1.start.z : 0.0;
line1.end.x : 0.0;
line1.end.y : 0.0;
line1.end.z : 30.0;
line2.length : 60.0;
line2.start.x : 0.0;
line2.start.y : 60.0;
line2.start.z : 0.0;
line2.end.x : 0.0;
line2.end.y : 0.0;
line2.end.z : 0.0;
line3.length : 10.0;
line3.start.x : 0.0;
line3.start.y : 0.0;
line3.start.z : 0.0;
line3.end.x : 10.0;
line3.end.y : 0.0;
line3.end.z : 0.0;
origin.x : 0.000000;
origin.y : 0.000000;
origin.z : 0.000000;

END_INSTANCE;

END_DATA;
```

Figure 83-85: Response File Created by the Interface Program

Finally (step 8), the response file is loaded back into the APM with the purpose of solving for the values still missing (such as `length1`, `critical_area`, `thickness`, `hole1.volume`, etc.) using the relations defined in the APM. Figure 83-86 is the solved data displayed by the APM Browser. Notice that, in this set of data, all attributes have value.

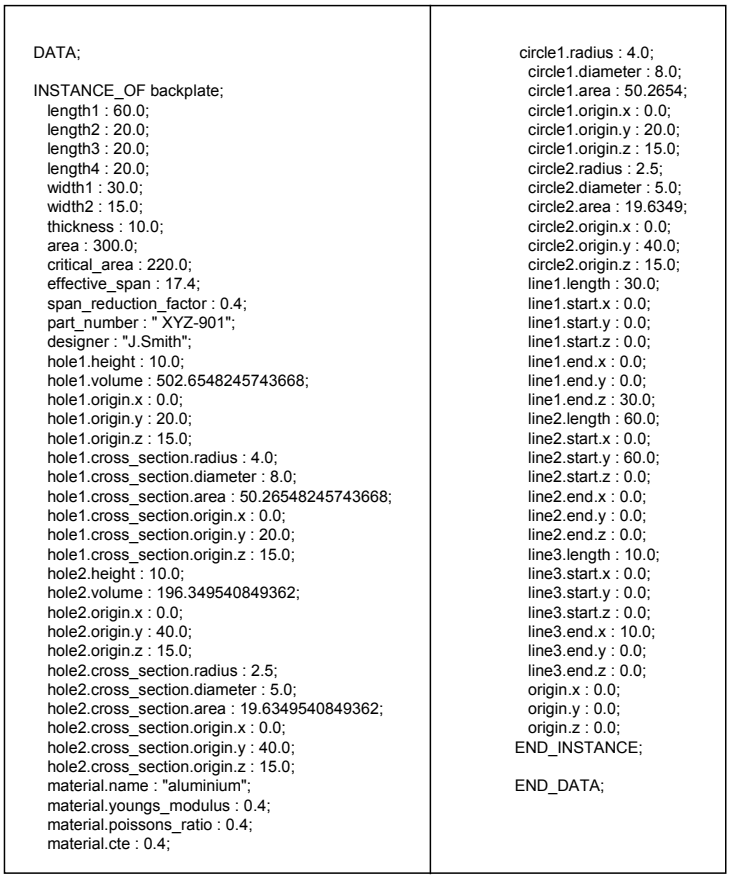


Figure 83-86: Solved Data Displayed by the APM Browser

APM-Design Tool Interface Test Using the Dimension-Tagging Technique

By the time of this writing, members of the EIS Lab were working on preliminary demonstrations of the second tagging approach described in Subsection 60; directly tagging the *dimensions* of interest instead of the *objects* in the solid model.

In the dimension-tagging approach, the dimensions of interest are tagged directly on the solid model with names that correspond to the names of the attributes in the APM (Figure 83-87). For example, the radius of the larger hole of the back plate is tagged in the solid model as "`hole1.cross_section.radius`", where `hole1`, `cross_section` and `radius` are the names defined in the APM. Hence, this approach eliminates the need of having to define additional geometric entities (such as circles and lines) or "artificial" attributes (such

as attributes `circle1`, `circle2`, `line1`, `line2`, and `line3` in domain `backplate`) in order to be able to connect the values of the geometric primitives being read from the solid model with the design features of the part.

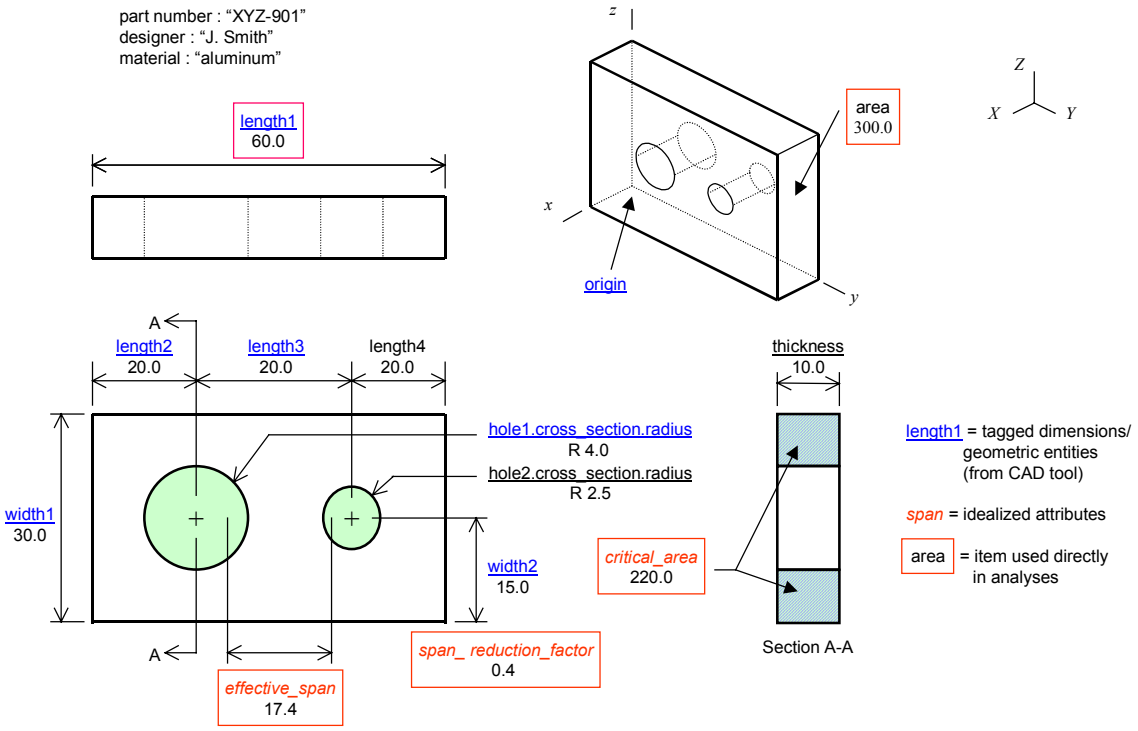


Figure 83-87: Dimension-Tagged Solid Model

Preliminary results show that this approach is better and most broadly applicable since it supports a wider variety of underlying geometries. However, typically only queries (output) are supported. A third approach, called "true parametric tagging", is also being investigated, which appears to offer greater capabilities, including both input and output, but at the cost of increased design creation complexity.

CHAPTER 7

EVALUATION

The objective of this chapter is to evaluate to what extent the APM Representation presented in Chapters 38, 64 and 83 meets the research objectives stated in Chapter 27. First, Section 98 establishes the scope of this evaluation. Next, Section 99 describes the evaluation approach used. Section 100 analyzes the results obtained and evaluates them against the research objectives. Section 109 briefly summarizes the results of this evaluation.

Evaluation Scope

The purpose of this evaluation is to assess the degree to which the APM Representation presented in this thesis satisfies the special requirements of design-analysis integration and, consequently, how adept this representation is when it comes to integrating design with analysis applications.

Recall from Section 40 that the APM Representation consists of the following four components:

- APM Information Model;
- APM Definition Languages;
- APM Graphical Representations; and
- APM Protocol.

Evaluating the APM Representation is equivalent to evaluating its four components against the research objectives. If at least one of these components meets a given objective then the conclusion is that the APM Representation meets the objective.

This evaluation did not address the following:

- The analysis models themselves and the validity of their results: essentially, the analysis applications used in the test cases of this thesis and the analysis models on which they are based are viewed as "black boxes" whose validity is not questioned in this thesis.
- The constraint solvers: many mature and reliable constraint solvers are available commercially and publicly. This thesis does not evaluate the efficiency or validity of their algorithms.
- The prototype implementation of the APM Protocol: the objective of developing a prototype implementation for this thesis was only to illustrate the various operations of the APM Protocol. The assumption was made that commercial software developers can develop more robust, efficient and elegant implementations of the APM Protocol than the one provided with this prototype.

Evaluation Approach

The objectives presented in Chapter 27 were used as a basis for the evaluation of the APM Representation. As stated in that chapter, these objectives represent the requirements of a suitable design-analysis integration representation. Therefore, the extent to which the APM Representation meets these objectives indicates how fit the APM Representation is as a design-analysis integration representation.

To aid with this evaluation, a table summarizing the results of the test cases such as the one shown in Figure 97-1 was constructed first. The rows of this table contain the thesis objectives listed in Chapter 27 and the columns contain the test cases developed in this thesis, grouped by Test APM Definitions and Test APM Client Applications. An additional group of columns labeled "Components" contains the four components of the APM Representation.

#	Thesis Objectives	Test APM Definitions				Test APM Applications					Components			
		Flap Link	Back Plate	Wing Flap Support	PWA	PWB Bending Analysis	Flap Link Extension Analysis	Back Plate Analysis/Synthesis	APM Browser	APM-Design Tool Interfacing	Information Model	Definition Languages	Graphical Reps	Protocol
1	Objective Text													

Figure 97-1: Evaluation Table Headings

Each objective was evaluated against each test case and the corresponding fields were filled with one of the following symbols:

"●" (a black dot): indicates that the test case successfully illustrated how the APM Representation meets the objective.

Blank: indicates that the test case did not attempt to illustrate how the APM Representation meets the objective or that the test case was not applicable to the objective.

It is important to stress the fact that, in this evaluation, a "●" for a given test case and objective only indicates that the test case provided a successful *example* of the APM Representation meeting the objective. As such, it could not be considered as conclusive evidence that the APM Representation meets the objective *in general*. On the other hand, blank fields only indicate that the test case did not address the objective or that the test case was not applicable to the objective, and therefore no conclusions were drawn from them either.

The final decision of whether or not the APM Representation meets a given objective was largely based on the results of the test cases but, ultimately, it was a subjective one. The test cases provided a good indication of whether or not an objective was met - especially if they are representative cases of the intended applications of the model - but, strictly speaking, they do not *prove* anything. As indicated by Sargent (Sargent, Tew et al. 1994), subjectively deciding if a model is valid based on the results of the various tests and evaluations is the most common decision-making approach for model validation used by model developers. A more quantitative evaluation using scores could have also been used but, as argued in this paper, these scores are also determined subjectively. Hence, the subjectiveness of this

approach tends to be hidden and thus it only *appears* to be objective. Furthermore, they also argue, a model could get a "passing score" and yet have defects that need correction.

This final decision of whether or not the APM Representation meets a given objective was recorded in the columns labeled "Components". For each objective, a check mark (✓) in one of these columns indicates that the objective was satisfactorily met by the corresponding APM component. A check mark followed by an asterisk (✓*) means that the objective was only met *partially* and that more work on meeting the objective is recommended. A blank field indicates that the APM component did not address the objective at all. At least one of these columns should have a check mark in order to conclude that the APM Representation meets the objective.

Chapter 110 will discuss the gaps evidenced by this evaluation and provide some recommendations for future work and possible extensions to attempt to fill them.

Evaluation of Results

Tables 97-1, 97-2 and 97-3 summarize the results of the evaluation. The reader is referred to Chapter 27 for the descriptions of each objective. What follows is an objective-by-objective analysis explaining each row of these tables in more detail.

#	Thesis Objectives	Test APM Definitions				Test APM Applications					Components			
		Flap Link	Back Plate	Wing Flap Support	PWA	PWB Bending Analysis	Flap Link Extension Analysis	Back Plate Analysis/Synthesis	APM Browser	APM-Design Tool Interfacing	Information Model	Definition Languages	Graphical Reps	Protocol
	Analysis-Oriented View Definition Objectives													
1	Constructs for defining analysis-oriented views	●	●	●	●						✓	✓	✓	
2	Bridge semantic gap between design and analysis	●	●	●	●	●				●	✓	✓	✓	✓
3	Creation of concise APMs	●	●	●	●	●	●	●	●	●	✓	✓	✓	
4	Easy APM creation, modification and extension	●	●	●	●						✓	✓	✓	
	Multiple Design Sources Support Objectives													
5	Support for multiple design sources	●	●	●	●	●	●	●	●	●	✓	✓	✓	✓
6	Representation of design data integration	●	●	●	●	●	●	●	●	●	✓*	✓*		✓*
	Idealization Representation Objectives													
7	Explicit representation of idealization knowledge	●	●	●	●	●	●	●	●	●	✓*	✓*	✓*	✓*
8	Definition of reusable idealizations	●	●	●	●		●				✓	✓	✓	✓
9	Definition of multi-level idealizations	●		●			●				✓*	✓	✓	✓
	Relation Representation and Constraint Solving Objectives													
10	Definition of complex relations	●	●	●	●	●	●	●	●	●	✓*	✓*	✓*	✓*
11	Definition of multidirectional relations	●	●	●	●	●	●	●	●	●	✓*	✓	✓	✓*

Table 97-1: APM Representation Evaluation Results (1 of 3)

#	Thesis Objectives	Test APM Definitions				Test APM Applications					Components			
		Flap Link	Back Plate	Wing Flap Support	PWA	PWB Bending Analysis	Flap Link Extension Analysis	Back Plate Analysis/Synthesis	APM Browser	APM-Design Tool Interfacing	Information Model	Definition Languages	Graphical Reps	Protocol
12	Dynamic relaxation of relations							●			✓			✓
13	Support for multiple constraint solvers										✓*			✓*
14	Constraint-solver independent relations	●	●	●	●						✓*	✓*	✓*	✓*
15	Easy definition and modification of relations	●	●	●	●	●	●	●	●	●	✓	✓	✓	✓
	Analysis Support Objectives													
16	Multiple analysis models and solution methods	●	●	●	●		●				✓	✓	✓	✓
17	Flexibility to add new analyses	●	●	●	●		●				✓	✓	✓	✓
	Data Access and Client Application Development Objectives													
18	Operations to access APM-defined information					●	●	●	●	●	✓			✓
19	Definition of late-bound operations					●	●	●	●	●				✓*
20	Reduce complexity of analysis code					●	●	●	●	●	✓	✓		✓
21	Isolate analysis applications from design data format					●	●	●	●	●	✓			✓
22	Constraint-solver independent client applications					●	●	●	●	●	✓	✓		✓
23	Hide constraint-solving details					●	●	●	●	●	✓	✓		✓

Table 97-2: APM Representation Evaluation Results (2 of 3)

#	Thesis Objectives	Test APM Definitions				Test APM Applications					Components			
		Flap Link	Back Plate	Wing Flap Support	PWA	PWB Bending Analysis	Flap Link Extension Analysis	Back Plate Analysis/Synthesis	APM Browser	APM-Design Tool Interfacing	Model	Definition Languages	Graphical Reps	Protocol
	Compatibility Objectives													
24	Leverage of existing product data standards				●	●					✓			✓
25	Support for multiple design formats	●	●	●	●	●	●	●	●	●	✓			✓
26	Compatibility with existing CAD/CAE tools	●	●				●			●	✓*	✓*		✓*
27	Compatibility with the MRA	●	●	●	●						✓	✓	✓	✓
	General Objectives													
28	Domain-independent	●	●	●	●	●	●	●	●	●	✓	✓	✓	✓
29	Unambiguous and formal definitions	●	●	●	●						✓			
30	Computer-interpretable form	●	●	●	●							✓		✓
31	Graphical forms	●	●	●	●								✓	
32	Provide correct results	●	●	●	●	●	●	●	●	●	✓	✓		✓

Table 97-3: APM Representation Evaluation Results (3 of 3)

Analysis-Oriented View Definition Objectives

Objective 1: *Provide the necessary constructs for defining analysis-oriented views of an engineering part.*

The APM Representation defines the basic building blocks for modeling analysis-oriented views of an engineering part. In this thesis, these views were called *Analyzable Product Models* (APMs). The constructs for defining APMs presented in this thesis are:

1. APMs;
2. APM Source Sets;
3. APM Domains;
4. APM Attributes (idealized and product);
5. APM Product Relations and APM Product Idealization Relations;
6. APM Source Set Link Definitions.

Each of these constructs was formally defined in Section 41 and mapped into EXPRESS in Section 65. The APM Structure Definition Language (APM-S) presented in Subsection 52 specified the syntax for defining each of these constructs. In addition, domains, attributes and relations can also be represented graphically using APM EXPRESS-G Diagrams (Subsection 55) and APM Constraint Schematics Diagrams (Subsection 56).

The utilization of these constructs was demonstrated with the definition of the APMs for a flap link (Subsection 85), a back plate (Subsection 86), a wing flap support (Subsection 87) and a PWA (Subsection 88). All these APM definitions showed how information from more than one design source was combined to obtain a single analyzable view of the product. They also showed how idealizations of different degrees of complexity can be defined.

The information defined by an APM can be utilized by more than one client application. This was demonstrated with the Flap Link Extension Analysis Application (Subsection 91), which combined two analysis applications into one: a formula-based and a finite element-based tension analysis. This example demonstrated how two analysis applications can share

design and idealized information defined in the same APM (the Flap Link APM of Subsection 85).

Objective 2: *Bridge the semantic gap between design and analysis representations.*

In the APM approach, analysis applications *drive* the development of APMs. Therefore, the structure, domains and attributes of these APMs are specifically modeled to meet the requirements of the analysis applications the APM is meant to support. As a result, APMs present design information at a semantic level more compatible with analysis models, thus effectively bridging the semantic gap between design and analysis.

It is important to note that APMs do not *eliminate* the semantic mismatch between design and analysis representations; design and analysis are inherently different and therefore this semantic mismatch is unavoidable and has to be dealt with at some point or another. What APMs do is add an additional layer of information between design and analysis, providing a stepping stone to perform the semantic translation before the actual utilization of this information by the analysis applications. As discussed in Subsection 60, the analyst still has to define the semantic mappings (for example, using a mapping language such as EXPRESS-X, as illustrated in Figure 38-58), but the APM provides means to represent these mappings more easily by explicitly defining the target representation for the mappings. To illustrate this, consider the example shown in Figure 97-2, where a semantic mapping between AP210 and an APM is being defined. This mapping specifies how to map the outline of a PWB from the source AP210 design representation to the target APM representation. Such mapping would have to be defined and executed by the analyst before loading the source set data into the APM.

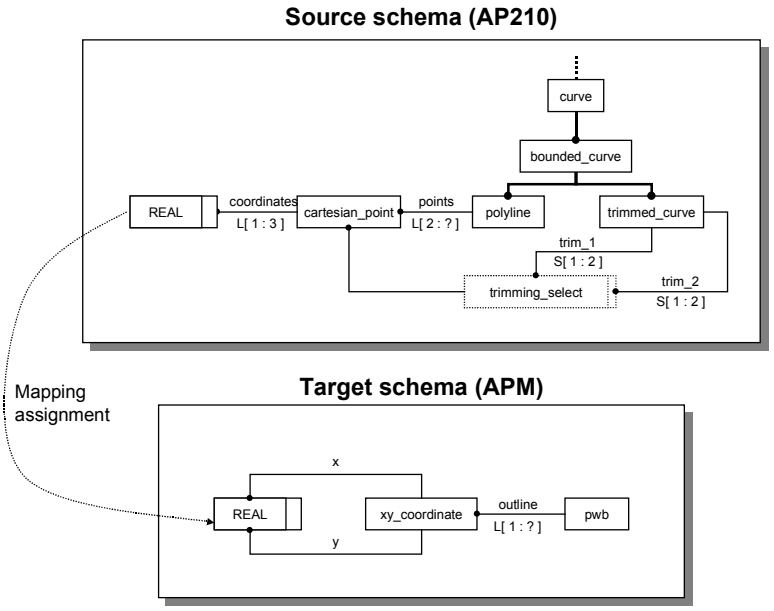


Figure 97-2: Semantic Mapping Definition Between AP210 and an APM

The semantic translation approach from design representations to analysis representations described in Subsection 60 was demonstrated in two of the test APM applications presented in Chapter 83. The first was the PWB Bending Analysis Application (Subsection 90). In this test case, one of the files used for the analysis (a STEP P21 file containing the geometry of the PWA) was obtained by translating a STEP AP210 file (Appendix A) into the format specified by the Printed Wiring Assembly APM (Subsection 88). There was a considerable semantic mismatch between the AP210 representation and the APM representation used for analysis that needed to be resolved during this translation (Tamburini, Peak et al. 1996; Tamburini, Peak et al. 1997). The second application in which the semantic mapping was demonstrated was in the APM-Design Tool Interfacing Tests (Section 94). In these tests, the design tool's API was used to perform the semantic translation between the representation of the design tool and the APM.

Objective 3: *Enable the creation of concise analyzable product models.*

The APM Representation allows the creation of APMs that contain just the right amount of information needed for a family of analyses. An APM with just one domain and one

attribute, for instance, would be perfectly valid if that is all the information that the analysis applications need.

Most of the APM Definitions presented in this thesis are relatively small and simple. However, the reduction of model size and complexity was particularly evidenced by the Printed Wiring Assembly APM (Subsection 88). The model from which this APM was populated in the PWB Bending Analysis Application (Subsection 90) was STEP AP210. STEP AP210 defines several hundred entities and a complex structure. In contrast, the corresponding analyzable view defined by the Printed Wiring Assembly APM is very simple and only contains a few dozen entities, yet it still satisfies the requirements of several analyses, including the PWB Bending Analysis of Subsection 90.

Objective 4: *Allow easy creation, modification and extension of analyzable product models.*

APMs are defined using the APM Structure Definition Language (APM-S) presented in Subsection 52. The syntax of APM-S is simple and contains relatively few constructs, and therefore domain experts should be able to model their own APMs with little assistance from data-modeling experts. Source sets, domains, attributes, relations and source set links can be easily created, deleted, modified or extended with an ordinary text editor. APM graphical representations (Section 54), which can be used as visual tools for developing, communicating, or documenting APMs, are equally easy to create and modify.

Multiple Design Sources Support Objectives

Objective 5*: Support for multiple sources of design information.*

The APM Representation provides a mechanism for explicitly defining how information from multiple design sources should be combined to create a single, unified analyzable view. This mechanism is based on the concepts of APM Source Sets and APM Source Set Links. APM Source Sets (Subsection 45) are a way to group APM Domains by design source. APM Source Set Links (Subsection 46) specify when and how instances from different source sets should be joined (or "linked") in order to obtain a single set of instances.

The APM Protocol specifies operations to load the instances for each source set and then link them according to the rules specified by the source set links. These operations are,

respectively, **`APMInterface.loadSourceSetData`** and **`APM.linkSourceSetData`** (Subsection 79). All the test cases presented in this thesis demonstrated the use of more than one design source.

Objective 6: *Allow explicit representation of design data integration knowledge.*

APM Source Sets (Subsection 45) and APM Source Set Links (Subsection 46) provide a formal, implementation-independent mechanism for explicitly representing how information from multiple design sources should be combined. The integration rules defined with APM Source Set Links are part of the APM itself and are independent from the specific implementation of the APM Protocol. This mechanism prevents analysis applications from having to include code to combine design information.

The APM Representation, however, places certain restrictions on how source set links can be defined. Recall from Subsection 46 that these restrictions limit the choice of attributes that can be selected as key, insertion, and inserted attributes in the definition of a source set link. The result of incorporating these restrictions in the more general definition of APM Source Set Links (Definition 38-57) resulted in the simplified definition of APM Source Set Links provided in Definition 38-59.

Recall also from Subsection 46 that one of the elements of an APM Source Set Link Definition is the *link condition*. This link condition is a boolean proposition that is evaluated to determine whether or not two instances from two source sets should be joined. As stated in that subsection, link conditions are limited in this work to a proposition of the form:

$key\_attribute_1$ *logical_operator* $key\_attribute_2$

Where *logical_operator* may be equal ("=="), greater than (">"), greater or equal than (">="), less that ("<"), less or equal than ("<="), or not equal ("!="). As it was discussed in that subsection, more complex link conditions could be defined between primitive values, which could involve if-then rules or arbitrary algorithmic procedures. The prototype implementation of the APM Protocol developed for this thesis limits this even further by supporting only the "equal" logical operator.

Although these restrictions simplify the definition and implementation of source set links, they also limit them to the simplest cases. Hence, as it will be recommended in Chapter 110,

the definition of APM Source Set Links and further implementations of the APM Protocol should be extended to accommodate more complex cases.

Source set links were graphically represented in several occasions Chapter 64 in conjunction with constraint schematics diagrams (Figures 38-63 and 64-15, for example). However, the nomenclature for representing source set links graphically was not formalized. A formal nomenclature for representing source set links would be valuable, but its development will also be left as a recommended future extension in Chapter 110.

Idealization Representation Objectives

Objective 7: *Allow explicit representation of idealization knowledge.*

Idealization knowledge is captured in the APM Representation through idealized attributes (Subsections 47, 52 and 67) and product idealization relations (Subsections 48, 52 and 71). The APM Representation provides the necessary constructs to define idealized attributes and product idealization relations as part of the APM itself.

Idealized attributes can also be represented graphically in APM EXPRESS-G and Constraint Schematic Diagrams. In EXPRESS-G (Subsection 55) idealized attributes are represented by adding the label "`(I)`" in front of the attribute name and in constraint schematic diagrams as shaded circles. Relations are represented in constraint schematic diagrams as labeled boxes. The label inside the box indicates the name of the relation. Free-form lines are used to link the related attributes to the relation.

All test cases developed in this thesis define one or more idealized attributes and their corresponding product idealization relations. For example, the Flap Link APM (Subsection 85) defines an idealized attribute called `effective_length` as one of the attributes of domain `flap_link`, and a product idealization relation `pir1` to derive its value from design attributes as follows:

```
DOMAIN flap_link;
   IDEALIZED effective_length : REAL;
   < other attributes …>
 PRODUCT_IDEALIZATION_RELATIONS
   pir1 : "<effective_length> == <sleeve_2.center.x> -
```

```
            <sleeve_1.center.x> - <sleeve_1.radius> -
<sleeve_2.radius>";
   < other relations …>

END_DOMAIN;
```

As it was stressed in Subsection 90, one of the main benefits of the APM approach is that values of idealized attributes are queried via the APM Protocol in the same way as the values of regular attributes, despite the fact that they have to be calculated at run time using one or more of the relations defined in the APM.

Another key aspect of the APM Representation – the ability to relate design features of a part to idealized features - was demonstrated in Subsection 87 (Wing Flap Support APM). In this test case, one of the features of the wing flap support (the bulkhead attachment point) was idealized as a generic channel fitting. By doing this, an analysis model available for channel fittings could be used to estimate the stresses in the bulkhead attachment point. As pointed out in that subsection, a significant amount of idealization knowledge – rarely documented or captured explicitly anywhere - is required to perform such associations between design and idealized features. As demonstrated in this test case, the APM Representation provides a mechanism to capture these associations explicitly.

Finally, the APM Representation provides the capability to define individual *attributes* of a given domain as idealized. In certain cases, however, it might be useful to define an entire *domain* as idealized. This would allow defining entire features or subparts of a part as idealized, adding semantic expressiveness to the model. Currently, this has to be done indirectly by declaring all the attributes of the domain as idealized. Chapter 110 will recommend extending the APM Representation to include the concept of idealized domains.

Objective 8*: Allow the definition of reusable idealizations*.

Idealized attributes and product idealization relations are defined as part of the APM, and therefore can be reused by any application accessing it.

All idealizations defined in the test cases of this thesis are potentially reusable. However, the only test case that demonstrated the reuse of idealizations was the Flap Link Extension Analysis Application (Subsection 91), as it was the only application that actually involved

more than one analysis. In this application, the idealized attribute `effective_length` was used by both the formula-based and the finite element-based analyses.

Objective 9: *Allow the definition of multi-fidelity idealizations.*

The APM Representation introduces the concept of multi-level domains (Subsection 42) for describing attributes at different levels of idealization fidelity. Different levels of a given idealized attribute are grouped into a multi-level domain, which is then used as the type of the idealized attribute. For example, in the Flap Link APM test case (Subsection 85) the multi-level domain `cross_section` was used to group two idealization levels for the critical cross section of the beam of the flap link (`detailed` and `simple`) as follows:

```
DOMAIN beam;
   critical_cross_section : MULTI_LEVEL cross_section;
   length : REAL;
   ESSENTIAL tf : REAL;
   ESSENTIAL tw : REAL;
   ESSENTIAL t2f : REAL;
   ESSENTIAL wf : REAL;
   ESSENTIAL hw : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
   pir5 : "<critical_cross_section.detailed.tf> == <tf>";
   pir6 : "<critical_cross_section.detailed.tw> == <tw>";
   pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
   pir8 : "<critical_cross_section.detailed.wf> == <wf>";
   pir9 : "<critical_cross_section.detailed.hw> == <hw>";
END_DOMAIN;
```

Where:

```
MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
 PRODUCT_IDEALIZATION_RELATIONS
   pir10 : "<detailed.wf> == <simple.wf>";
   pir11 : "<detailed.hw> == <simple.hw>";
   pir12 : "<detailed.tf> == <simple.tf>";
   pir13 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;
```

Once a multi-level domain is defined, product idealization relations can then be used to specify how the attributes of the different levels of the domain are related to each other. For

example, relation `pir5` above specifies that attribute `tf` of the `detailed` level corresponds to the "real" `tf` of the beam. Later, relation `pir12` specifies that the `tf` of the `simple` level is equal to the `tf` of the `detailed` level.

Multi-level domains can also be represented in APM EXPRESS-G (Subsection 55) and APM Constraint Schematics Diagrams (Subsection 56). Multi-level domains are represented in EXPRESS-G diagrams using a box with a diagonal line in its upper-left corner and in APM Constraint Schematic Diagrams using a diamond symbol. The levels of the multi-level domains are represented in the same way as regular attributes in both diagrams.

The APM Protocol allows APM client applications easy access to any level of a multi-level domain. For example, the detailed cross section of an instance of domain `flap_link` called **`flapLinkInstance`** would be accessed with:

```
flapLinkInstance.beam.cross_section.detailed
```

and the simple cross section with:

```
flapLinkInstance.beam.cross_section.simple
```

The Flap Link APM (Subsection 85) and the Wing Flap Support APM (Subsection 87) demonstrated the use of APM Multi-Level Domains. The Flap Link Extension Analysis Application (Subsection 91) demonstrated an analysis in which the user could actually choose at run time between two levels of idealization (simple and detailed) for the critical cross section of the flap link.

Relation Representation and Constraint Solving Objectives

Objective 10: *Allow the definition of complex relations.*

Currently, the APM Representation allows the definition of closed-form mathematical relations that can be captured with a single string. These expressions may include algebraic operations (addition, subtraction, multiplication and division), transcendental functions (trigonometric, exponential, logarithmic, etc.), powers, and a limited set of aggregate operations (sums, averages, minimums, maximums). Relations can be also represented graphically using APM Constraint Schematics Diagrams (Subsection 56) and APM Constraint Network Diagrams (Subsection 57).

The APM Representation, however, does not provide a way to define relations that require conditional statements, iterations or calls to external procedures. Development of a more expressive syntax for specifying relations will be recommended in Chapter 110.

Admittedly, the specification presented in this work remains vague as to what is considered a valid expression in a relation. The mathematical operations and symbols that can be used in an expression, for instance, are not formally specified. A more detailed specification of this syntax will be also recommended in Chapter 110. This syntax should be generic enough to allow the utilization of any constraint solver (**Objective 14**), while at the same time extensible to take advantage of the more powerful capabilities of some constraint solvers. For this work, the syntax for defining expressions was informally limited to the standard algebraic operations (+, - , * , /) and the aggregate operations (SUM, MAX, MIN, AVG) defined in Subsection 48.

As discussed in Subsection 88, one limitation of the current prototype implementation of the APM Representation is that it does not support *relation overriding*. In other words, it is not possible to replace a relation that is being inherited from a parent domain with a different one. As pointed out in that subsection, this leads to unnecessary replication of relations that could have been otherwise inherited from a common parent domain if one or more subtypes need to redefine inherited relations. Support for relations overloading will be recommended in Chapter 110 for future implementation releases.

Objective 11: *Allow the definition of multidirectional relations.*

In the APM Representation, the way relations are defined does not imply any particular input/output combination of the attributes that participate in them. Which attributes are inputs and which are outputs is determined at run time. For example, the following relation:

```
r1 : "<length> == <l1> + <l2>";
```

could also have been defined in any of the following ways:

```
r1 : "<length> - <l1> - <l2> == 0";
r1 : "<length> - <l1> == <l2>";
r1 : "<length> - <l2> == <l1>";
r1 : "0 == <length> - <l1> - <l2>";
r1 : "<l2> == <length> - <l1>";
r1 : "<l1> == <length> - <l2>";
```

to yield the same results.

In order to enable this dynamic definition of input/output combinations **APMRealInstance**s have a boolean attribute called **isInput** (Subsections 68 and 81). This attribute is set to **true** at run time when the instance is an input in a given relation and to **false** if it is an output. As explained in Subsection 81, this attribute is used by the constraint-solving algorithms to determine which values need to be solved by the constraint solver.

By default, all the attributes whose values have been populated in the design sources are initially declared as inputs. However, this can be changed later by the user, if desired. For this purpose, the APM Protocol provides methods **APMRealInstance.setAsInput** and **APMRealInstance.setAsOutput** (Subsection 81). These methods can be used to declare an **APMRealInstance** as an input or as an output, respectively. As discussed in Subsection 81, these methods take into account the effect that changing an instance from input to output (or vice versa) has on the rest of the instances in the constraint network.

Multiple input/output combinations were tested in the test cases of this thesis by adding or removing values directly in the source data files and re-loading the data. In addition, The Back Plate Analysis and Synthesis Application (Subsection 92) demonstrated the capability of the APM Representation to support synthesis as well as design by letting the user dynamically change the input/output directions of the relations defined in the Back Plate APM (Subsection 86).

However, as it was pointed out in Subsection 81, the APM Representation does not specify any mechanism to prevent the user from determining invalid input/output combinations. As a consequence, the user can potentially specify invalid input/output combinations that lead to conflicting solutions or to no solutions at all. As discussed in that subsection, a more sophisticated version of method **APMRealInstance.setAsInput** could inspect the constraint network and automatically flag other variables as outputs as the user declares the inputs, thus avoiding invalid input/output combinations. Adding this additional capability will be recommended as an extension for this thesis in Chapter 110.

In addition, there might be some relations that do not allow multiple input/output directions, or that become ambiguous or discontinuous when used in a particular direction.

For example, the following relation defining a constraint between the length, width and diagonal distance of an PWB:

```
r : "<diagonal_length>*<diagonal_length> == <width>*<width> +
<length>*<length>";
```

would yield two solutions (a positive and a negative) for any given two inputs. In such cases, it may be sufficient to define additional constraints (such as constraints specifying that all lengths must be positive) to eliminate the ambiguity. In addition, there might be cases in which relations contain calls to external routines (although this is not yet supported by the APM Representation) that are difficult or impossible to reverse. For example, consider the following relation:

```
r : "<fitting_factor> == CURVE_FIT( a , b , c , d )";
```

This relation calls the external routine `CURVE_FIT` with parameters `a`, `b`, `c` and `d` as inputs to obtain `fitting_factor`. In this case, it may be difficult (or even impossible) to reverse the direction of the relation (say, for example, to give `fitting_factor`, `a`, `b` and `c` as inputs to obtain `d`). This may be either because CURVE_FIT is a "black box" (does not allow changes to the combinations of inputs and outputs) or because its algorithm does not allow reversion. In such cases, it may be useful to be able to restrict the possible input/output directions of the relation so that only valid directions can be used. The `ONEWAY` construct introduced by the COB language (an extension of the APM-S language - discussed in Section 113) partially addresses this problem by declaring the variable in the left-hand-side of a relation as the output, thus allowing only one input/output combination. Hence, the relation above would be rewritten as:

```
r : ONEWAY : "<fitting_factor> == CURVE_FIT( a , b , c , d )";
```

Even with this type of restricted relations it might still be possible to handle different input/output combinations by using some kind of iterative approach. For example, in the example pointed out above in which `fitting_factor`, `a`, `b` and `c` are given as inputs to obtain `d`, the relation might be run iteratively in its only direction allowed until a value of `d` that yields the `fitting_factor` specified is found.

Objective 12: *Allow dynamic relaxation of relations.*

The APM Representation supports dynamic relaxation of relations. For this purpose, APM Constraint Network Relations (Subsections 50 and 72) have an attribute called `active`, which is a boolean flag used to indicate whether a relation is active (when its value is true) or relaxed (when its value is false). As discussed in Subsection 81, relaxing a relation effectively removes it from the constraint network. As a consequence, relaxed relations are ignored during the process of building the systems of equations that are going to be sent to the constraint solver to try to find the value of an unknown attribute.

The APM Protocol defines a method for dynamically inactivating or "relaxing" a relation in the constraint network (method **`ConstraintNetworkRelation.setActive`** – Subsection 81). The use of this method was demonstrated in the Back Plate Analysis and Synthesis Application (Subsection 92). In this application the user was able to relax or activate a relation by checking or unchecking a checkbox next to the relation.

Objective 13: *Support for multiple constraint solvers.*

The APM Representation introduces the concept of APM Solver Wrappers (Subsection 81) to provide a mechanism for isolating the APM Protocol from the choice of constraint solver. Class **`APMSolverWrapper`** is an APM abstract class from which wrappers for specific solvers (such as **`MathematicaWrapper`**) are subtyped. The **`APMSolverWrapper`** class defines the functionality that these wrappers must implement in order to handle the communication between the constraint solvers and the APM. APM Solver Wrappers receive a request from the method **`APMRealInstance.trySolveForValue`** (Subsection 81) get value operations to solve a system of equations, translate these requests into the appropriate solver-specific commands, run the solver, get the results, and send them back in a neutral form specified in advance.

This wrapping approach provides a modular architecture that facilitates replacing one constraint solver with a different one. Replacing one constraint solver with another will only reflect in the choice of **`APMSolverWrapper`** used. For example, the following statement contained in method **`APMRealInstance.trySolveForValue`** creates an **`APMSolverWrapper`** for the case in which Mathematica is used as the constraint solver:

```
APMSolverWrapper solver =
   APMSolverWrapperFactory.makeSolverWrapperFor( "mathematica" );
```

If another constraint solver were used, the only change required would be in the argument of function **`APMSolverFactory.makeSolverWrapperFor`** above. For example, if Maple were now used as the new constraint solver the line above would change to:

```
APMSolverWrapper solver =
   APMSolverWrapperFactory.makeSolverWrapperFor( "maple" );
```

The remaining code of method **`APMRealInstance.trySolveForValue`** remains intact, since the APM Solver Wrappers handle all the constraint solver-specific details. Of course, any application code using the APM Protocol also remains unaffected.

The test cases in this thesis utilized Mathematica as the only constraint solver. Unfortunately, no attempt was made to replace Mathematica with another constraint solver to demonstrate the claims made above. Such test will be recommended for future work in Chapter 110.

Objective 14: *Allow constraint solver-independent definition of relations.*

The definition of constraint solver-independent relations is enabled by two elements of the APM Representation: Constraint Networks (Subsections 50 and 72) and the APM Constraint-Solving Technique (Subsection 81).

During the process of parsing and loading the definition of an APM (Subsection 78), a constraint network is built directly from the relations defined in the APM. This constraint network represents attributes and relations as a "flat" network of interconnected nodes, making it easier to determine which attributes and relations are connected to a given attribute or relation. As described in Subsection 81, this representation is used by the constraint-solving technique to determine which relations should be shipped to the constraint solver in order to find the unknown value of a requested attribute. With these relations, special objects known as APM Solver Wrappers build a system of equations using the syntax of the specific constraint solver being used. They then execute the appropriate solving routines, get the results back from the solver, put these results in terms of `apm_solver_results` (Subsection 75) and send them back to the APM get value operations. This constraint-solving approach makes it possible to specify a generic syntax for

APM relations and then let the APM Solver Wrappers handle the translation into the specific syntax of the constraint solvers.

However, as discovered with the test cases of this thesis, a syntactic translation such as the one performed by the APM Solver Wrappers is sometimes not sufficient, since some constraint solvers may have limitations that need to be taken into account. For example, a given constraint solver may not support systems with non-linear equations. Other - more idiosyncratic - limitations may prove more to be difficult to identify or predict. For example, in the test cases Mathematica behaved in an unexpected way if a relation had an unknown attribute in the denominator of a term (it would return no solutions even if there were). In these cases, the relation had to be rearranged to remove the offending attribute from the denominator before sending it to Mathematica. In this thesis, this rearrangement of relations to accommodate Mathematica was performed manually, by redefining the relations directly in the APM definition. Consequently, the relations defined in the test APMs of this thesis are not strictly independent from the constraint solver.

The ability to modify a relation to overcome some limitation or idiosyncrasy of the constraint solver could prove to be very difficult – in some cases even impossible - to implement. In this thesis, it was possible to work around the lack of such capability since only one constraint solver (Mathematica) was being tested. However, more work is needed in defining a syntax for the APM relations that is generic enough to be utilizable by multiple constraint solvers, while at the same time extensible to be able to take advantage of the more powerful capabilities of some constraint solvers.

Such a generic syntax would require some level of translation in the wrapper (not attempted in this thesis). For example, the generic syntax could specify that x^2 should be used to express x2. However, a particular constraint solver could use x**2 instead. Then the wrapper should translate the relation containing x^2 into x**2 to accommodate the syntax of the particular constraint solver.

Objective 15: *Allow easy definition and modification of relations.*

Relations can be easily modified by editing their corresponding mathematical expressions defined in the APM Definition File using an ordinary text editor. No changes are required in the codes of the applications that use that APM definition: the new relations take effect

automatically the next time the new APM definition is read into the application. Changes would be needed, however, if new attributes need to be displayed on the graphical user interface or existing attributes are deleted.

Analysis Support Objectives

Objective 16: *Allow support for multiple analysis models and solution methods.*

From the point of view of an APM, different analysis models and solution methods translate into different information requirements. For example, a finite-element analysis may require more detailed geometric information about a part that a simpler, less accurate formula-based solution. Hence, adding support for additional analysis models or solution methods to an APM is generally a matter of adding new domains, attributes and relations to the model if needed. As more analysis models and solution methods are added to the picture, the potential for information sharing increases and less new information has to be added to the APM.

The ability to define multi-fidelity idealizations (**Objective 9**) is also important in meeting this objective. As discussed in Subsection 42, multi-level domains can be used for describing attributes at different levels of idealization fidelity. This is useful when two analysis models or solution methods call for the same information but at different levels of accuracy.

The Flap Link Extension Analysis Application (Subsection 91) demonstrated a case in which two solution methods (formula- and finite element-based) are supported. In this example, the finite-element analysis required more detailed information about the geometry of the flap link that the formula-based analysis.

Objective 17: *Provide flexibility to add additional analyses.*

This capability is addressed in **Objective 16**.

Data Access and Client Application Development Objectives

Objective 18: *Provide a programming language-independent specification for analyzable product model data-access operations.*

Together, the APM Information Model (Section 41) and the APM Protocol (Section 58) specify the minimal set of classes, variables and operations that must be provided by any given implementation of the APM Representation. These implementations can be delivered as libraries of APM classes in specific programming languages that application developers may use to develop their APM-driven applications. Chapter 64 presented a prototype implementation of the APM Representation in the Java programming language and Chapter 83 demonstrated its utilization with a few test cases.

Even though this specification of classes and operations is programming-language independent, it relies on object-oriented concepts such as inheritance and polymorphism and therefore is easier to implement using object-oriented programming languages. However, there are programming techniques that can be used to map object-oriented concepts into non-object-oriented languages (Rumbaugh, Blaha et al. 1991).

Sections 58 and 77 discussed in greater detail some operations of the APM Protocol that were considered critical to the APM Representation (APM Definitions Loading, Source Set Data Loading, APM Data Usage and APM Data Saving).

Objective 19: *Allow the definition of late-bound operations.*

One of the advantages of defining a generic, domain-independent APM Information Model was that it allowed the specification and implementation of late-bound operations; indeed, all the operations specified in the APM Protocol are late-bound. Late-bound operations enable the development of APM-generic applications that access and manipulate data defined by any APM. The APM Browser developed in this thesis (Subsection 93) provided an example of a generic application demonstrating the utilization of these late-bound operations.

Alternatively, it is possible to develop APM-Specific (or non-generic) applications using early-bound operations as well. For example, an application that always uses the Flap Link APM of Subsection 85 could potentially use the following hypothetical statement to get the

value of attribute `effective_length` from an instance of domain `flap_link` called **`flapLinkInstance`**:

```
L = flapLinkInstance.getEffectiveLength();
```

instead of its equivalent late-bound expression:

```
L = flapLinkInstance.getRealInstance( "effective_length"
   ).getRealValue();
```

The first statement (early-bound) is clearly simpler and will probably execute faster that the second (late-bound). In the late-bound approach method **`getRealInstance`** must check (at run time) whether or not the `flap_link` domain contains indeed an attribute called `effective_length` and then method **`getRealValue`** must check if this attribute is a real number. In the early-bound approach, this check could be performed at compile time. The obvious disadvantage is that a specialized function called **`getEffectiveLength`** would have to be created to get the value of attribute `effective_length`, and this function would only work on instances of `flap_link`. Similar "get" functions (and their "put" counterparts) would have to be created for each attribute of each domain that is going to be accessed. Although it is possible to create these functions automatically from the APM definition, they would involve more than just returning or setting the value of a given class variable. These early-bound functions would also have to support the multidirectional nature of APM relations. Hence, the early-bound approach would also require some constraint-solving technique similar to the one implemented in this thesis for the late-bound approach. Consequently, the early-bound function **`getEffectiveLength`** will probably end up wrapping methods **`getRealInstance`** and **`getRealValue`** as follows (method **`getRealInstance`** could probably be modified so that it does not check the name of the attribute):

```
double getEffectiveLength {

   return this.getRealInstance( "effective_length" ).getRealValue();

}
```

The early-bound approach has one obvious major disadvantage: it cannot be used to write generic applications. For example, an APM browser such as the one developed for this thesis (Subsection 93) could not have been written using early-bound operations, since the

structures of the APMs that are going to be loaded into the application are not known in advance.

Hence, the ideal approach is to make the two approaches available. Developers could then write early-bound applications (also known as "class-based" or, in APM terms "domain-based") when the structure of the APM is known in advance, thus benefiting from a simpler syntax and better performance. Alternatively, they could use the late-bound approach when there is the need to write "generic" applications that work with any APM. This thesis only specifies and implements the late-bound approach. However, as it will be recommended in Chapter 110, it would be of great value to develop a compiler that generates early-bound operations for specific APMs.

Objective 20: *Reduce the complexity of analysis code.*

The APM absorbs much of the complexity that would be otherwise passed to analysis applications. The result is much simpler and easier to maintain analysis applications. The APM approach relieves analysis applications from having to perform the following tasks:

- Parsing and translating design data;
- Resolving the semantic mismatch between design and analysis representations;
- Combining design information from multiple sources;
- Transforming design information to obtain idealized information;
- Handling multiple input/output combinations of idealized and design data;
- Handling the constraint solving details triggered when the value of an unknown attribute is queried.

Objective 21: *Isolate analysis applications from the format of the design data.*

APM client applications utilize method **`APMInterface.loadSourceSetData`** to load APM-defined design data. The only arguments that need to be passed to this method are the names of the files where the source set data is stored. Once this method is performed, the APM data is available to the application in the form of APM Domain Instances. No code is

required in the client application to perform any data-formatting conversion; the data-formatting details are entirely hidden from the application within method **`APMInterface.loadSourceSetData`**.

The Source Data Wrapping Technique discussed in Subsections 60 and 79 enables this isolation of the data-formatting details from the APM applications. As discussed in Subsection 79, method **`APMInterface.loadSourceSetData`** uses the services of special objects called APM Source Data Wrapper Objects, which "know" the formatting details of the design data and provide a neutral communication mechanism between format-specific data parsers and the APM. These objects are instances of some subtype of class **`APMSourceDataWrapperObject`**; and there is one subclass of this class for each data format supported. This thesis demonstrated the utilization of two types of source set data wrapper objects: one to read STEP data (**`StepWrapper`**) and another to read APM-I data (**`APMInstanceWrapper`**).

Objective 22: *Allow development of constraint solver-independent client applications.*

APM client applications are not "aware" that a constraint solver is being used underneath to solve for the unknown values that they request. In other words, APM client applications never deal with the constraint solver directly. In addition, the choice of constraint solver is completely hidden from the code of the APM application: if a constraint solver is replaced with a different one, the code of the APM application remains unaffected.

From the point of view of the APM client application, the method that ends up triggering the constraint-solving request is **`APMRealInstance.getRealValue`**. As explained in Subsection 81, if the APM Real Instance does not have value then method **`APMRealInstance.getRealValue`** will call method **`APMRealInstance.trySolveFor-Value`** which in turn builds the constraint system, sends it to the constraint solver, and gets the results back. All this activity between this method and the constraint solver is hidden from the APM client application.

Objective 23: *Hide constraint-solving details from client applications.*

This capability is addressed in **Objective 22**.

Compatibility Objectives

Objective 24: *Leverage existing product data exchange standards.*

The APM Representation provides support for multiple design data formats (**Objective 25**) including, of course, *standard* formats such as STEP P21.

If in addition to the *format* of the data the *structure* is also standard, there is the added advantage that the semantic mapping described in Subsection 60 to solve the semantic mismatch between design and analysis representations becomes tool-independent. For example, recall the example of Figure 38-58, in which a solid modeler stores its data in STEP AP203 format. In this example, a STEP mapping language such as EXPRESS-X (Spooner, Hardwick et al. 1996) could be used to map AP203 data to the APM model. This mapping is between a standard schema (AP203) and the APM, and therefore is independent of the tool that creates the data. If the tool is replaced with a different one, the same EXPRESS-X mapping definitions can be reused as long as the new tool conforms to AP203.

A semantic mapping between a standard data exchange format and the APM was demonstrated in the PWB Bending Analysis Application (Subsection 90). In this test case, one of the design data files had been originally created with an E/CAD tool (Mentor Graphics) in STEP AP210 format and translated into APM format with a mapping program developed by the author using STEP Tools Inc.'s ST-Developer Toolkit (STEP Tools Inc 1997c).

Objective 25: *Support multiple design data formats.*

The APM Representation introduced a technique called *Source Data Wrapping* to facilitate the loading of design data stored in multiple data formats into the APM (Subsections 60 and 79). In this technique, objects called APM Source Set Data Wrapping objects provide a neutral communication mechanism between format-specific data parsers and the APM. Format-specific data wrappers (such as the **`StepWrapper`** and the **`APMInstanceWrapper`** developed in this thesis to parse STEP P21 and APM-I data, respectively) are subtyped from class **`APMSourceDataWrapperObject`** and deal with the formatting details of the source set data. They parse the source set data, perform the necessary conversions, and pass it to the APM load source set data operation in terms of format-independent objects called APM

Source Set Data Wrapper Returned Values (Subsection 74). The messages and the information exchanged between the APM source set data loading operation and the specific data wrapper are independent from the data format being read. Adding support for an additional data format only involves adding a new subtype of **`APMSourceDataWrapperObject`** for the specific format.

The test cases in this thesis demonstrated the use of two design data formats: an APM-native format called APM-I (Subsection 53) and the STEP P21 format.

Objective 26: *Compatible with existing CAD/CAE Tools*.

Being compatible with existing *CAD* tools requires being able to bridge the semantic mismatch between design and analysis representations and support multiple design data formats. Hence, this part of the objective is addressed by Objectives **2**, **24** and **25**. On the other hand, being compatible with existing *CAE* tools requires that these tools be able to access the design and idealized information defined in an APM. This access could be either direct or through other analysis programs that use the CAE tools as solution engines. This part of the objective is addressed by **Objective 18**.

Two alternative design model "tagging" techniques – termed *object tagging* and *dimension tagging* - were proposed in Subsection 60 to facilitate the semantic translation between the native representations of the design tools and the APM Representation. A couple of APM-design tool interfacing tests (described in Section 94) demonstrated how a model created with a commercial solid modeling system (Dassault Systemes' CATIA) was loaded into an APM using each one of these tagging approaches.

However, as pointed out in that section, the exact mechanism through which a model is tagged will eventually depend on the capabilities of the specific design application and its API, as well as on how the semantic translator is actually implemented. As it will be recommended in Chapter 110, more work is still needed on formalizing this tagging approach and clearly specifying general requirements for CAD tools to make them compatible with the APM approach.

The PWB Bending Analysis Application described in Subsection 90 demonstrated the compatibility of the APM Representation with a design tool through the utilization of

standard data exchange format. In this test case, the source data file containing the geometry of the PWA used for the test case had been originally created with Mentor Graphic's ECAD application and translated into STEP AP 210.

On the other hand, the Flap Link Extension Analysis Application (Subsection 91) demonstrated a case in which a commercial finite-element program (Ansys) was used as a solution engine by an APM client application utilizing the APM protocol.

All test cases also demonstrated how the APM interfaced with Wolfram Research's Mathematica (Wolfram 1996) which, in the context of this discussion, can be viewed as an analysis tool using information contained in the APM.

Objective 27: *Compatible with the Multi-Representation Architecture (MRA).*

The APM Representation is compatible with the MRA approach developed at the Georgia Institute of Technology by Drs. Russell S. Peak and Robert E. Fulton (see Subsection 9). As illustrated in Figure 97-3, the APM Representation complements the MRA by providing the product information required by "Product Model", thus filling the gap between design tools and PBAMs. Originally, the Product Model was assumed to be equivalent to semantically rich product models like STEP AP210. However, the multi-fidelity idealization nature of analysis leads to an insatiable information appetite that no product model, no matter how rich, can continually satisfy. Thus, the APM technique is necessary as a general link to design tools in order to harmonize diverse data and add idealizations and missing data.

Together, the APM Representation and the MRA provide a highly modular and flexible design-analysis architecture. However, the APM Representation does not depend on the MRA in order to be utilizable. In fact, it is possible to use the APM Representation in conjunction with analysis models that do not explicitly conform to the MRA.

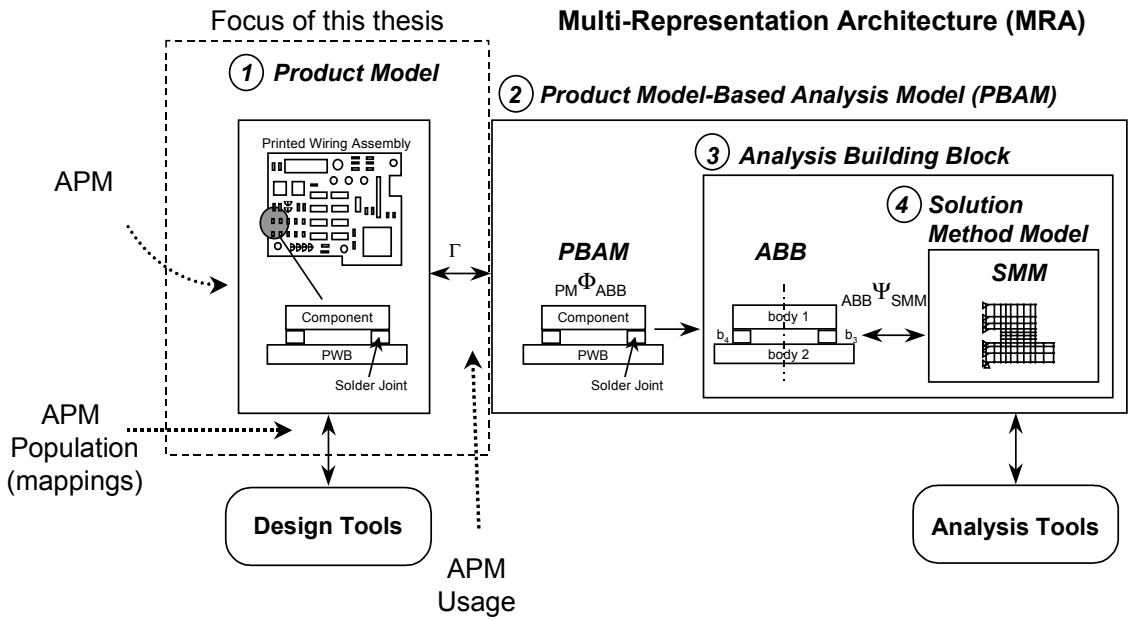


Figure 97-3: APM-MRA Compatibility

Depending on the programming languages involved (the language in which the APM Protocol is implemented and the language in which PBAMs are written), PBAMs may directly access the information contained in the APM through the APM Protocol, just as any other APM client application would do. Alternatively, a separate APM client application could be written that extracts the information needed and puts it in a format that PBAMs can read[67]. More recent work at the EIS Lab (Section 113) demonstrates this compatibility.

General Objectives

Objective 28: *Be product domain-independent.*

The entire APM Representation is based on generic concepts such as source sets, domains, attributes, relations, source set links and domain instances. These concepts are independent from any particular domain application and therefore can be used to describe virtually any domain. As a result, the APM Representation effectively provides a "template" for creating domain-specific APMs.

[67] This latter approach was used in the Tiger project to perform PWB bending and solder-joint deformation analyses (see Subsection 11).

The Test APM Definitions developed in this thesis demonstrate how the APM Representation can be used to define APMs for different domains. For example, the Wing Flap Support APM (Subsection 87) belongs to the airplane structures domain, whereas the Printed Wiring Assembly APM (Subsection 88) belongs to the electronic systems domain.

The APM Protocol also helps support this domain independence with the concept of late-bound operations (**Objective 19**).

Objective 29: *Provide unambiguous and formal definitions.*

The fundamental constructs of the APM Representation were formally defined in Section 41 and collectively presented as the APM Information Model - the first of the four components of the APM Representation. These conceptual building blocks provide a theoretical foundation for the APM Representation. Their definitions were built upon set theory concepts and expressed in mathematical terms, making them unambiguous and independent from any particular data modeling or programming language. These definitions served as the basis for the subsequent development of the other three components of the APM Representation: the APM Definition Languages (Section 51), the APM Graphical Representations (Section 54) and the APM Protocol (Section 58).

Objective 30: *Have a computer-interpretable form.*

The APM Representation provides a computer-interpretable language for defining APMs. The syntax of this language, called the APM Structure Definition Language (APM-S), was defined in Section 52. The tokens and grammars that make up the APM-S language were provided in Appendix D. These tokens and grammars were used to write the specification files for the lexer and parser generation utilities (Jlex and Java CUP, respectively) used in this work (Subsection 78). These specification files were included in Appendices E and F. A Java lexer and parser were developed in this thesis using these utilities to scan and parse APM definitions and create the appropriate APM instances in memory that can be accessed and manipulated by analysis applications.

Objective 31: *Have some type(s) of graphical form(s).*

The APM Representation specifies three graphical representations that can be used as visual tools for developing, communicating, or documenting APMs. Each representation conveys a particular aspect of the APM better than the others, and therefore they should be used in a complementary fashion. These representations are:

1. APM EXPRESS-G Diagrams (Subsection 55): useful for representing the object structure (domains, attributes, is-a hierarchy) of the analyzable model.
2. APM Constraint Schematics Diagrams (Subsection 56): useful for representing part-of domain hierarchies and the relations between attributes.
3. APM Constraint Network Diagrams (Subsection 57): useful for representing constraint networks and showing how attributes are interconnected through relations (in a flattened way, as opposed to constraint schematics which also show relations but stress the part-of hierarchy).

Of these three graphical representations, only the APM Constraint Network Diagrams were entirely original to this work. The other two were extensions or adaptations of existing graphical representations.

Objective 32: *Provide correct results.*

The results of each Test APM Definition and each Test APM Application developed for this thesis were individually checked for correctness. This validation was performed by manually verifying that the values of all derived and idealized attributes were consistent with the relations defined in their corresponding APMs, and that the values of the design values coming from the design sources had not been altered. The APM Browser was extensively used for this purpose, since it provided the additional advantage over the Test APM Applications of displaying the values of *all* the attributes defined in an APM instead of only those used for a particular analysis. In addition, the results from the Wing Flap Support

APM case study (Subsection 87) compared well with the original Strength Check Note[68] (Figure 83-36) in which they were calculated by traditional means.

Evaluation Summary

This chapter evaluated the APM Representation presented in Chapter 38 against the objectives for design-analysis representations defined in Chapter 27. The implementation presented in Chapter 64 and results of the case studies presented in Chapter 83 were used as a basis for this evaluation.

This evaluation evidenced the strengths and weaknesses of the APM Representation and recognized some unfulfilled gaps that need some additional work. Need for additional work was identified in areas of the APM Representation such as representation of design data integration (**Objective 6**), integration with CAD/CAE tools (**Objective 26**), representation of idealization knowledge (**Objective 7**) and definition of APM relations (Objectives **10**, **11**, and **14**).

Despite these gaps, this evaluation indicates that the APM Representation satisfactorily meets most of the objectives defined in Chapter 27. Many of the reasons for not fully meeting some of the objectives appear not be fundamental deficiencies; therefore, it is expected that with additional time and work these issues can be overcome. In fact, this work is currently being used in industrial projects and significantly extended by other members of the research group of which the author was a member (Chandrasekhar 1999; Wilson 1999; Wilson, Peak et al. 1999), and some of the limitations identified here are being overcome as this thesis is being written (see Subsection 113). Therefore, the conclusion of this evaluation is that the APM Representation meets the overall objective of this thesis: developing a useful product model view for design-analysis integration.

[68] Strength Check Notes are documents that provide a detailed report of the analyses performed for a particular part of an airplane. They are used for internal reference at typical aerospace companies as well as for reporting to external regulatory agencies.

CHAPTER 8

RECOMMENDED EXTENSIONS

"Art is never finished, only abandoned."

(Leonardo DaVinci)

This chapter proposes several extensions to this work aimed at overcoming the limitations and unfulfilled objectives identified throughout this thesis. Section 113 describes some extensions and refinements that - by the time of this writing - were being implemented by the research team at the Engineering Information Laboratory at Georgia Tech.

What follows is a list of recommended extensions to this thesis grouped into two groups: extensions that are likely to require further research (Subsection 111) and extensions that involve adding new features or capabilities to APM implementations (Subsection 112).

Extensions Requiring Further Research

1. Take advantage of the mathematical framework defined in this thesis to study the mathematical aspects of the APM Representation and define interesting properties and theorems that could be derived from it (Section 63).
2. Formalize the design model tagging approach used to enable the semantic translation from design representations to the APM representation. Specify clear requirements for CAD tools to ensure that they are compatible with the APM approach (Subsection 60).
3. Provide a clear, unambiguous, constraint solver-independent syntax for defining APM Relations. Specify the mathematical operations and symbols that can be used in a

relation. This syntax should be generic enough to allow the utilization of any constraint solver while at the same time extensible to take advantage of the more powerful capabilities of some constraint solvers (Subsection 52).

5. Perform a more formal analysis to assess the efficiency of the algorithms presented in this thesis and refine them as necessary. It is important that these algorithms be as efficient as possible, as some of the APM operations are likely to be computationally intensive. Of particular importance are the algorithms for critical operations such as loading the analyzable product model definitions, loading and combining the design data, and coordinating the constraint-solving process required to calculate the values of the idealized attributes. The only significant attempt made in this thesis to improve the efficiency of an algorithm was in method **`APMRealInstance.trySolveForValue`**. As discussed in Subsection 80, when building the system of equations that is going to be sent to the constraint solver, this method only uses the relations that are *connected* to the variable whose value is unknown as opposed to *all* the relations defined in the APM. This, of course, reduces the size of the system of equations sent to the constraint solver therefore reducing execution time. However, more work is needed to improve the efficiency of this and the other algorithms presented in this thesis. For example, when **`APMRealInstance.trySolveForValue`** builds the system of constraints to solve for a variable, it includes *all* the relations connected to the unknown variable. This, however, may not be necessary. For example, consider the simple constraint network of Figure 110-1 and assume that the operation is trying to find the value of variable **a** given the values of variables **b** and **c**. Also assume that **R1** and **R2** are linear, algebraic constraints. In this case, it is not necessary to include relation **R2** in the system of constraints to solve for **a**, even though **R2** is connected to it through the constraint network. Relation **R1** and the values of variables **b** and **c** will be sufficient for calculating the value of **a**. Of course, this conclusion was easy to reach in this case because the constraint network of this example is extremely simple. In more complex and interconnected constraint networks, and/or in constraint networks involving non-linear relations, finding out which relations can be excluded from the system of constraints may prove to be a much more difficult task. Another aspect of

this operation that could be improved is that currently, this operation has "no memory". To illustrate this, consider the same constraint network of the example above. Assume that in a first pass, the operation is required to find the value of variable **a** given the values of **c** and **d**. In order to solve for **a**, a system of two constraints (**R1** and **R2**) with two unknowns (**a** and **b**) is built. The solution of this system will yield values for **a** and - as a by-product - for **b** as well. If the value of **b** is queried later in another pass, it should not be necessary to send the same system of constraints to the constraint solver to solve for it. However, the current algorithm does not take advantage of this fact and therefore repeats the constraint-solving job this time looking for the value of **b**. For large constraint systems with many unknowns, this represents a considerable waste of computational time. Therefore, the efficiency of this operation could be significantly improved if the solutions of previous constraint-solving passes are stored for future constraint-solving passes (Subsection 81).

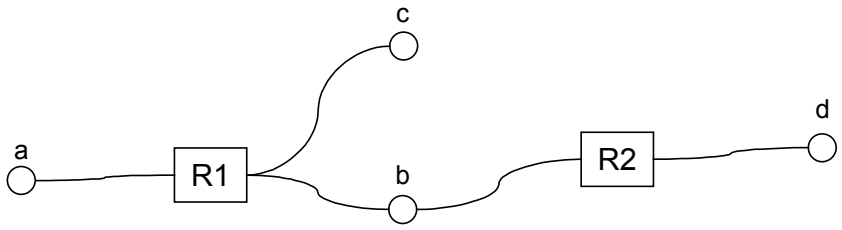


Figure 110-1: Constraint Network Example

4. Extend the APM Representation to include Design Requirements and Objectives (DR&O) information about the part. This type of information – which includes allowable stresses and deformations for different operating conditions - is normally provided at analysis time in order to calculate margins of safety. However, since it may be argued that this information actually *belongs* to the part, it could potentially be provided as part of the APM.
5. The current schema-based constraint network generation does not allow changes of the relations and/or the participating attributes once the constraint network is created. An interesting alternate approach would be to have instance-based constraint

networks. Such an approach would allow dynamic manipulation of the constraint network after it is created and support such things as higher order relations (for example, if **a** = **b** then **c** = **d** + **e**, else **c** = **a** + **b**) or relations between members of aggregates that grow dynamically.

6. The mechanism for defining idealizations in this thesis is based on "primitive" relations, that is, relations that explicitly involve specific primitive attributes of a given product. For example, the critical area of the back plate test case of Subsection 86 is defined as:

   pir_1 : "<critical_area> == (<width> - <hole1.diameter>) * <thickness>";

   However, it would be useful to be able to define "canonical" idealizations expressed in higher semantic terms that can be applied to a family of products. For example, the critical area idealization relation could be expressed as:

   pir_1 : "<critical_area> == <smallest_cross_section>" ;

   Such an approach would require some kind of geometric reasoning mechanism (to figure out which is the smallest cross section of the part) that could require the use of Artificial Intelligence techniques.

Extensions to APM Implementations

1. Extend the syntax for defining APM Source Set Links to support more complex cases than the currently supported. Provide more flexibility in the choice of key, insertion and inserted attributes in a source set link as well as the ability to declare arbitrary source set link conditions that involve more complex conditional or algorithmic expressions (Subsection 46).
2. Define a formal graphical representation for APM Source Set Links or extend the current nomenclature for APM Constraint Schematics Diagrams to support APM Source Set Links (Subsection 56).

3. Extend the APM Representation to allow the definition of entire idealized *domains*, in addition to idealized *attributes* only (Subsection 47).
4. Extend the syntax for defining APM Relations to enable the definition of more complex relations that include if-then statements, loops, iterations, conditionals and calls to external procedures. Add the possibility of defining relations between *complex objects* in addition to only between primitive attributes as currently defined (Subsection 48).
5. Refine operation **`APMRealInstance.setAsInput`** so that it inspects the constraint network and automatically flags other variables as outputs as the user declares the inputs. This way, the operation would prevent the user from specifying invalid input/output combinations that lead to conflicting solutions or to no solutions at all (Subsection 81).
6. Support overloading of APM Relations: in other words, allow the replacement or redefinition of a relation that is being inherited from a parent domain. This is actually only an extension for the current APM Protocol prototype implementation, since the APM Representation already allows redefinition of relations (Subsection 88).
7. Provide some mechanism to handle relations that do not allow multiple input/output directions, or that become ambiguous or discontinuous when used in a particular direction. This could involve adding additional constructs the APM Representation (such as the ONEWAY construct used in the COB language - discussed in Section 113) and/or improving the constraint solving algorithm to automatically use an iterative approach when a relation cannot be reversed.
8. Extend the APM Protocol to include the specification of early-bound APM-access operations in addition to the late-bound already specified. This extension will also require development of an APM compiler that generates these early-bound operations for specific APMs. This way, developers would have the choice of developing early-bound applications when the structure of the APM is known in advance (thus benefiting from a simpler syntax and better performance), or they could use the late-

bound approach when there is the need to write "generic" applications that work with any APM (Sections 58 and 76).

9. Perform additional experiments to verify the claims of constraint-solver independence made in this thesis. Replace the constraint solver used in the test cases of this thesis (Mathematica) with a different one and assess the impact of this change (Subsection 81).
10. Develop a toolkit for APM development and testing. This toolkit could include graphical tools for visually create, debug and edit APMs based on one or more of the graphical forms presented in this thesis. Other candidate tools for this toolkit are Graphical APM Browsers, APM Integrated Development Environments, and APM Conformance-Checking Tools.
11. Provide an UML version of the APM Information Model (Booch, Jacobson et al. 1998; Fowler 1998). This would provide an alternate representation of the APM Information Model and would facilitate communication and dissemination of the APM concepts to the currently growing UML audience. In addition, it could provide opportunities to leverage the increasing number of analysis, design and programming UML tools available.
12. Leverage the capability of current Product Data Management (PDM) systems to enable configuration control and change management of both the product and the evolution of its derived APM models.
13. Build libraries of reusable APM building blocks (such as the `channel_fitting` example of subsection 87).

Current Design-Analysis Research at The Engineering Information Systems Laboratory

During the writing of this thesis, the team at the EIS Lab has continued refining and extending the various design-analysis integration concepts, methodologies and tools - including the APM Representation presented in this thesis. The individual research efforts of

the members of the laboratory have been better integrated under the umbrella of what is now called the "*Extended*" Multi-Representation Architecture (Peak, Fulton et al. 1999). New concepts (or, more accurately, *evolutions* of existing ones) such as Constrained Objects (COBs) and Context-based Analysis Models (CBAMs) have been introduced. This section summarizes these newer developments and provides references to the latest reports and publications (some of which are still work in progress) produced by the EIS Lab team.

The MRA continues to provide the general framework for the various design-analysis integration components developed at the EIS Lab. As shown in Figure 110-2 the architecture is fundamentally the same as the one originally introduced by Peak (Peak, Fulton et al. 1998) and discussed in Subsection Figure 7-1, with the only difference that PBAMs have been replaced by CBAMs.

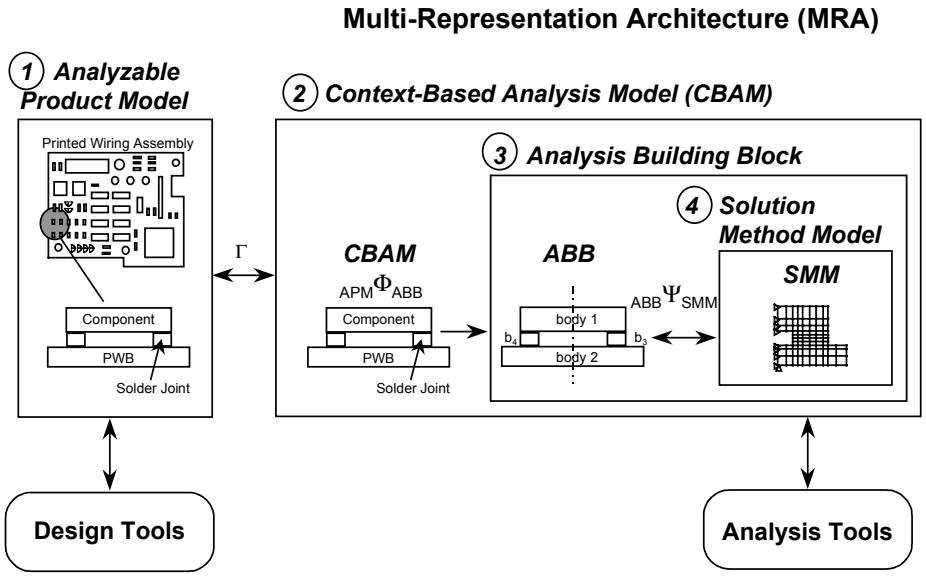


Figure 110-2: Extended Multi-Representation Architecture (MRA)

CBAMs (Figure 110-3) generalize the original PBAMs by adding associativity with the context of the analysis being done. This analysis context includes boundary condition objects (such as loads, conditions, links to next-higher analyses), behavior mode being analyzed and design objectives (such as margins of safety). An example of a CBAM for the extension

analysis of a linkage is shown in Figure 110-4. Another example – a CBAM for the analysis of channel fittings – was shown in Figure 83-41.

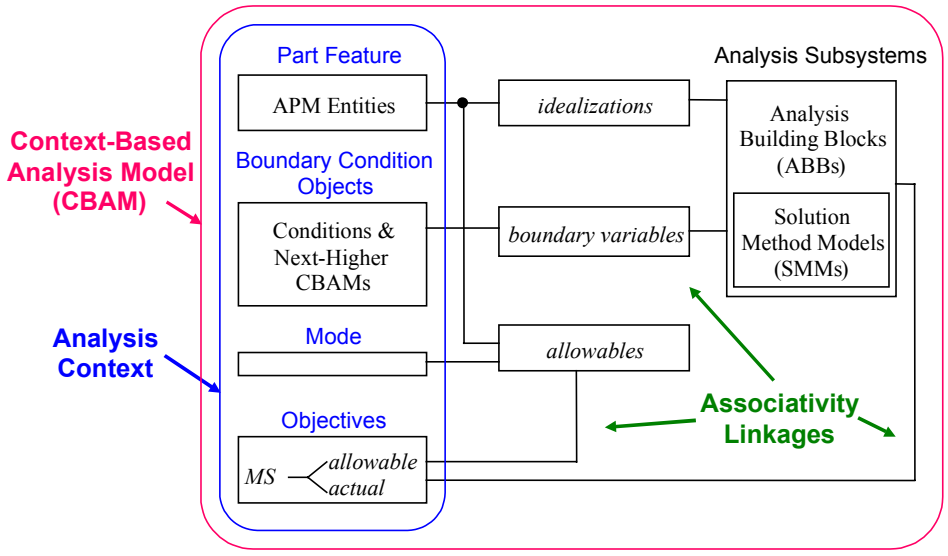


Figure 110-3: Structure of a Context-Based Analysis Model (CBAM)

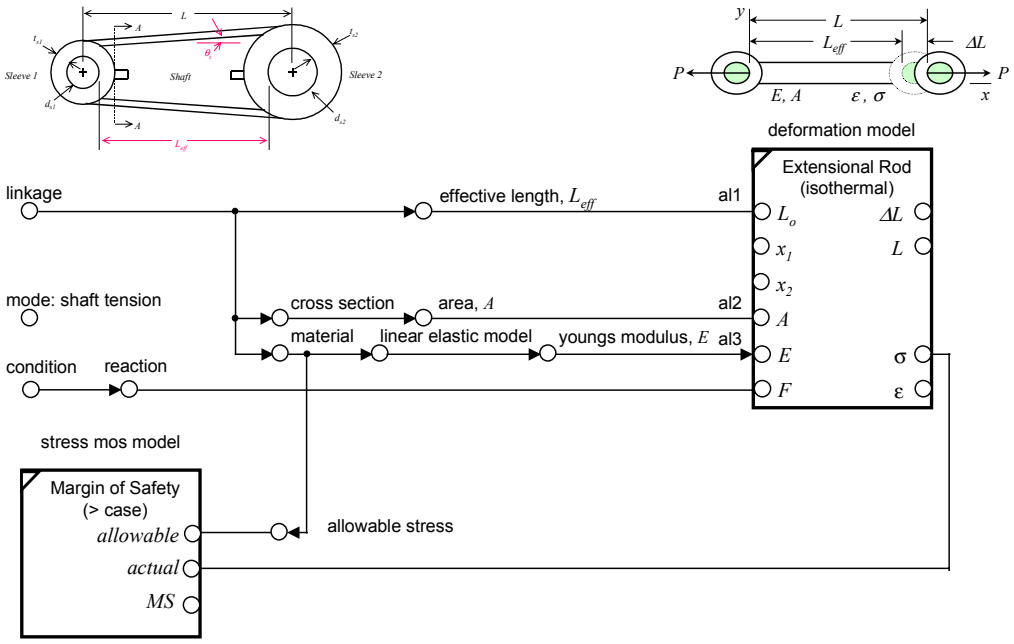


Figure 110-4: Linkage Extension Analysis CBAM

Another significant new development has been the *Constrained Object (COB) language*. The COB language is a generalization of the APM-S language presented in this thesis (Subsection 52) that allows the definition of any type of constrained object (objects whose attributes are related by mathematical constraints). With this generalization, the COB language can be used not only to define APMs, but also ABBs and CBAMs. Hence, the COB language now forms the basis of the extended MRA. The syntax of COB - described in detail in (Wilson 1999) and (Wilson, Peak et al. 1999) - is very similar to the syntax of APM-S. Essentially, only the names of some APM-S keywords that were too APM-Specific were replaced with more general names (for example, the `DOMAIN` keyword is now called `COB`) and a few new constructs and keywords were added. One of the new keywords added is `ONEWAY`, which allows the definition of unidirectional relations: when a relation is declared `ONEWAY`, the attribute in its left-hand-side will always be the output. The COB language also introduces the new construct `USE_FROM`, which allows reusing COB definitions from other schemas into the current schema. Thanks to this construct, COBs can be grouped into self-contained libraries and reused in many schemas to build other COBs. For example, generic analysis models (ABBs) could be grouped into libraries and used to build CBAMs. Figure 110-5 is an example of a COB schema for a spring system ABB. In this schema, COB `spring_system` uses COB `spring`, which was defined in another schema (`abbs.cos`) being included with the `USE_FROM` construct.

```
SCHEMA spring_system;

SOURCE_SET one ROOT_COB abb;

COB spring_system SUBTYPE_OF analysis_system;
   spring1 : spring;
   spring2 : spring;
   deformation1, u<sub>1</sub> : REAL;
   deformation2, u<sub>2</sub> : REAL;
   load, P : REAL;
 RELATIONS
   r1 : "<spring1.start> == 0.0";
   r2 : "<spring1.end0> == <spring2.start>";
   r3 : "<spring1.force> == <spring2.force>";
   r4 : "<spring2.force> == <load>";
   r5 : "<deformation1> == <spring1.total_elongation>";
   r6 : "<deformation2> == <spring2.total_elongation> +
                           <deformation1>";
END_COB;

USE_FROM lib/abbs.cos;

END_SOURCE_SET;

END_SCHEMA;
```

```
/* Note:
The following COB spring is used by spring_system and is
contained in lib/abbs.cos.

COB spring;
                undeformed_length, L<sub>o</sub> : REAL;
                spring_constant, k : REAL;
                start, x<sub>1</sub> : REAL;
                end0, x<sub>2</sub> : REAL;
                length, L : REAL;
                total_elongation, &Delta;L : REAL;
                force, F : REAL;
RELATIONS
   r1 : "<length> == <end0> - <start>";
   r2 : "<total_elongation> == <length> - <undeformed_length>";
   r3 : "<force> == <spring_constant>*<total_elongation>";
END_COB;
*/
```

Figure 110-5: COB Lexical Form for Spring System

The earlier MRA prototype implementation toolkit (DaiTools - discussed in Subsection 11) has also undergone significant improvement and is now called *XaiTools* (Wilson, Peak et al. 1999). XaiTools is a Java-based toolkit for X-analysis integration[69] that is a reference implementation of MRA concepts.

One of the tools included in this toolkit - the *COB Browser* – is currently the main user tool for browsing, executing and interacting with COBs. Figure 110-6 shows a screen shot of the COB Browser displaying an instance of the spring system of Figure 110-5. The COB Browser is similar in principle and scope to the APM Browser introduced in this thesis (Subsection 93), with significant improvements in usability and presentation.

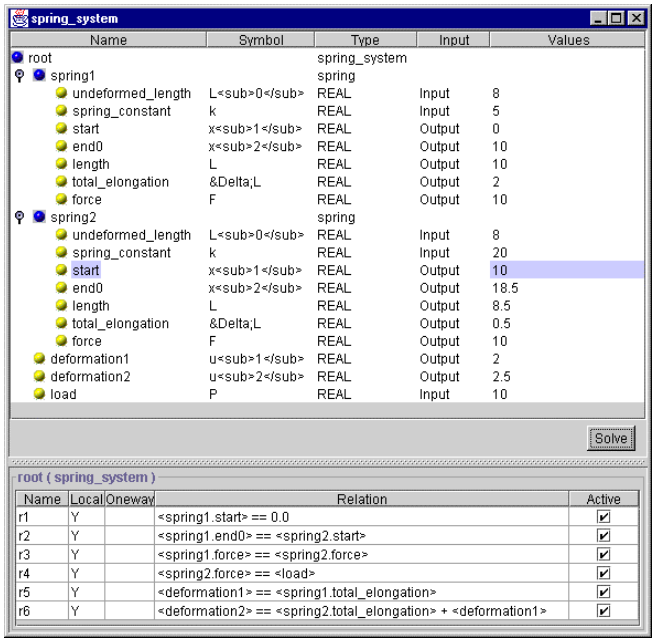


Figure 110-6: COB Browser

This more recent work demonstrates how others are utilizing the APM Representation (for example, creating new APMs) and building upon its conceptual foundation.

[69] Where X stands for design, manufacturing, etc.

CHAPTER 9

CONCLUDING REMARKS

This thesis has introduced a new representation of engineering products - termed Analyzable Product Model (APM) - aimed at facilitating design-analysis integration. This representation defines formal, generic, computer-interpretable constructs to create and manipulate analysis-oriented views of engineering parts or products. These analysis-oriented views combine design information from multiple design representations, add idealized information, and bridge the syntactic and semantic gap that exists between design and analysis representations, thus providing a unified perspective of the product that is more suitable for analysis and that can be shared by multiple analysis applications.

The APM Representation was formally introduced in Chapter 38. For presentation purposes, the APM Representation was divided into the following four components:

1. ***APM Information Model*** (Section 41): which provides the theoretical foundation of the APM Representation. It defines the basic constructs to build APMs. The definitions of these constructs were presented in mathematical form and therefore are independent from any particular data modeling or programming language. Among the basic constructs defined in the APM Representation are: APM Conceptual Building Blocks include APM Source Sets, APM Source Set Links, APM Domains, APM Product Attributes, APM Idealized Attributes, APM Product Relations, and APM Product Idealization Relations.
2. ***APM Definition Languages*** (Section 51): two APM Definition Languages were introduced in this work: the APM Structure Definition Language (APM-S), used to define the *structure* (that is, the source sets, domains, attributes, relations, and source set links) of specific APMs, and the APM Instance Definition Language (APM-I), used to define *instances* of the domains defined in these APMs.

3. ***APM Graphical Representations*** (Section 54): three APM Graphical Representations were defined: APM EXPRESS-G Diagrams for representing the structure (domains, attributes, is-a hierarchy) of an APM, APM Constraint Schematics Diagrams for representing domain part-of hierarchies and the relations between its attributes, and APM Constraint Network Diagrams for representing flattened constraint networks and showing how attributes are interconnected through relations.
4. ***APM Protocol*** (Section 58): defines the conceptual algorithms of the representation. It also provides a programming language-independent description of the operations that should be supported by specific implementations of the APM Representation. These operations may be used by developers of APM-driven applications to access APM-defined information.

APMs provide a stepping stone between design and analysis representations not previously available in other approaches. Having this intermediate representation allows separating the *creation* of the analyzable view of a product (which involves parsing, translating, integrating and idealizing design information) from its eventual *utilization* by analysis applications. This way, APMs absorb much of the complexity that would be otherwise passed to analysis applications, resulting in leaner and easier to maintain analysis applications, with the added advantage than APMs are simple to define and modify. The APM approach relieves analysis applications from having to perform the following tasks:

- Parsing and translating design data;
- Resolving the semantic mismatch between design and analysis representations;
- Combining design information from multiple sources;
- Transforming design information to obtain idealized information;
- Handling multiple input/output combinations of idealized and design data;
- Handling the constraint solving details triggered when the value of an unknown attribute is queried.

Two types of end users are expected to be the most benefited from the APM Representation: *designers/analysts* (often the same person) and *integrators* (the developers

of APM client applications). Some of the expected benefits for each of these two types of users are summarized in Table 114-1:

	Typical Scenario	Improved APM Scenario
Designer/Analyst		
Design Changes	Analysis model updated manually	Analysis model updated automatically
Analysis Results	Design model updated manually	Design model updated automatically
Analysis Audits	Manually trace analysis values back to design models (filling gaps/ guessing)	Idealizations provide explicit audit trail
Integrator		
Value Queries	- Must determine if and what relations need to be used. - If relations change, code changes	Constraint solving details handled behind the scenes. No constraint solving code needed. No code changes needed if relations change.
Analysis Code	Write it from scratch to support new analyses	Reuse other APMs. Extend only as necessary

Table 114-1: End User Benefits of Using the APM Representation

For designers and analysts the benefits are:

- *Design changes updated automatically:* in the typical design-analysis scenario (that is, without using the APM approach), if there is a change in the design the user has to manually update the analysis model. In the improved APM scenario, the analysis model is automatically updated thanks to the relations defined in the APM.
- *Automatic synthesis of analysis results:* in the typical scenario, the analyst runs the analyses and manually transforms their results into the design. In the APM scenario analysis results can be synthesized into design values automatically.

- *Explicit analysis audit trails:* in the typical scenario, auditing an analysis involves manually fill the gaps and a considerable amount of guessing in order to trace analysis values back to the design model(s) from where they originate. In the APM scenario, explicit idealization relations provide this missing audit trail.

For integrators the benefits are:

- *Simplified value queries:* ultimately, the purpose of an APM is to provide the values of the attributes (idealized or not) required by engineering analysis. For the developers of analysis applications, one of the most significant advantages of using the APM approach is that any of these values can be queried in the same way, regardless of whether it comes directly from the design repositories or has to be calculated at run time using one or more of the relations defined in the APM. The analysis application does not have to check first whether the attribute has value or not, nor figure out what relations are needed to calculate it. Moreover, if any of the relations defined in the APM is changed, the code of the application does not have to be modified to reflect this change, since the updated relations will automatically be used by the constraint solver in the next run. This thesis also introduced a mechanism to allow the user to logically group relations and attributes in meaningful semantic units - the APM Domains - rather than having to manage the entire set of relations as a whole. This approach gives him or her more control over the constraint solution process.
- *New analyses are easier to add:* in the traditional scenario, adding a new analysis often involves writing new code from scratch (sometimes with some code reutilization, at best). In the APM scenario, the integrator can reuse other APMs and extend them only as necessary.

A prototype implementation of this APM Representation was presented in Chapter 64. This prototype implementation was used in several test cases drawn from real applications from the electronic packaging and aerospace industries (described in Chapter 83) which helped test and validate the APM Representation. These test cases utilized several commercial CAD/CAE tools and STEP data exchange standards.

Among the characteristics of this APM Representation that are key to facilitating design-analysis integration are:

- Support for multiple sources of design data;
- Explicit definition of the rules required to combine these multiple sources of design data and derive single, analysis-oriented views of the product;
- Definition of idealized information and the transformations required to derive it from the design information;
- Support for multi-directional relations among idealized and design attributes;
- A generic programming interface that can be used to develop analysis applications that access APM-defined information;
- Independence from any particular design data formats, constraint solver or programming language.

Perhaps one of the most significant contributions of the APM Representation is that it provides a mechanism to explicitly define the relations between the design and idealized attributes of a part. This type of knowledge is key to design-analysis integration and crucial to achieving repeatable, traceable and reliable analyses, yet it is rarely contained in today's analysis documentation or explicitly captured anywhere. By providing the mechanism to formally describe the link between design and analysis representations, the APM Representation makes it easier to reproduce the idealization decisions made by the analyst and automate the idealization process.

REFERENCES

Al-Timimi, K. and J. MacKrell (1996). STEP: Towards Open Systems. CIMData Inc., Ann Arbor, MI.

Arabshahi, S., D. C. Barton, et al. (1991). "Towards Integrated Design and Analysis." Finite Elements in Analysis and Design 9(4): 271-293.

Arabshahi, S., D. C. Barton, et al. (1993). "Steps Towards CAD-FEA Integration." Engineering with Computers 9(1): 17-26.

Armstrong, C. G. (1994). "Modelling Requirements for Finite-Element Analysis." Computer-Aided Design 26(7): 573-578.

Assal, H. and C. Eastman (1995). "Engineering Database as a Medium for Translation." Conference on Building Models, Stanford, CA.

Bjorn, N. F.-B. and A. Borning (1992). "Integrating Constraints with an Object-Oriented Language." Proceedings of the 1992 European Conference on Object-Oriented Programming: 268-286.

Booch, G., I. Jacobson, et al. (1998). The Unified Modeling Language User Guide. Addison-Wesley Publishing Company.

Brooke, D. M., A. Pennington, et al. (1995). "An Ontology for Engineering Analyses." Engineering with Computers(11): 36-45.

Borning, A. and B. Freeman-Benson (1998). "Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics." Constraints: An International Journal 3(1): 1-26.

Borning, A., K. Marriott, et al. (1997). "Solving Linear Arithmetic Constraints for User Interface Applications." 1997 ACM Symposium on User Interface Software and Technology.

Chandrasekhar, A. (expected 1999). "Integrating Analyzable Product Models with Geometric CAD Models." Masters Thesis, George W. Woodruff School of Mechanical Engineering. Georgia Institute of Technology, Atlanta, GA.

Cimtalay, S., R. S. Peak, et al. (1996). "Optimization of Solder Joint Fatigue Life Using Product Model-Based Analysis Models." ASME Intl. Mech. Engr. Congress & Expo, Application of CAE/CAD to Electronic Systems, EEP-Vol.18.

Cutkosky, M. R., R. S. Engelmore, et al. (1993). "PACT: An Experiment in Integrating Concurrent Engineering Systems." Computer 26(1): 28-37.

Deitel , H. M. and P. J. Deitel (1998). Java: How to Program. Prentice Hall, Upper Saddle River, New Jersey.

Deitz, D. (1996). Java: A New Tool for Engineering. Mechanical Engineering**:** 68-72.

Deitz, D. (1997). "The Convergence of Design and Analysis." Mechanical Engineering 119(3): 93-100.

Denno, P. (1997). "NIST Expresso." Internet WWW page, at URL: http://www.mel.nist.gov/msidstaff/denno/nist-expresso.html (version current at 04/08/99).

Dragan, R. V. (1997). Java Tools Get Real. PC Magazine**:** 181-214.

Eastman, C. M. (1996). "Managing Integrity in Design Information Flows." Computer-Aided Design 28(6/7): 551-565.

Eastman, C. M. and N. Fereshetian (1994). "Information Models for Use in Product Design: a Comparison." Computer-Aided Design 26(7): 551-572.

Eastman, C. M., A. H. Bond, et al. (1991). "A Formal Approach for Product Model Information." Research in Engineering Design 2(4): 65-80.

Eastman, C. M., M. S. Cho, et al. (1995). "A Data Model and Database Supporting Integrity Management." ASCE International Computing Congress, Atlanta, GA.

Eastman, C., H. Assal, et al. (1995). "Structure of a Product Database Supporting Model Evolution." Conference on Building Models, Stanford, CA.

Elliot, B. (1997). "JLex: A Lexical Analyzer Generator for Java." Internet WWW page, at URL: www.cs.princeton.edu/~appel/modern/java/JLex/manual.html (version current at 08/12/1998).

Elmqvist, H. and S. E. Mattsson (1997). "An Introduction to the Physical Modeling Language Modelica." ESS '97 European Simulation Symposium, Passau, Germany.

Elmqvist, H., S. E. Mattsson, et al. (1998a). "Modelica - An International Effort to Design an Object-Oriented Modeling Language." Summer Computer Simulation Conference '98, Reno, Nevada.

Elmqvist, H., S. E. Mattsson, et al. (1998b). "Modelica - The New Object-Oriented Modeling Language." The 12th European Simulation Multiconference, ESM '98, Manchester, UK.

ESPRIT (1993). "Generic Analysis Model." Internet WWW page, at URL: www.newcastle.research.ec.org/esp-syn/text/8894.html (version current at 12/29/98).

ESPRIT (1996). "GEM Introduction." Internet WWW page, at URL: www.tno.nl/instit/bouw/project/gem/introduction/introduction.html (version current at 12/29/98).

Finn, D. P. (1993). "A physical modeling assistant for the preliminary stages of finite element analysis." (AI EDAM) Artificial Intelligence for Engineering Design, Analysis and Manufacturing 7(4): 275-286.

Finn, D. P., J. B. Grimson, et al. (1992). An Intelligent Modelling Assistant for Preliminary Analysis in Design. Artificial Intelligence in Design '92. J. S. Gero. Kluwer Academic Publishers, Netherlands**:** 597-596.

Fisher, C. N. and L. R. J. (1988). Crafting a Compiler. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA.

Flanagan, D. (1997). Java in a Nutshell. O' Reilly and Associates, Inc., Sebastopol, CA.

Fowler, M. (1998). UML Distilled : Applying the Standard Object Modeling Language. Addison-Wesley Publishing Company.

Fritzson, P. and V. Engelson (1998). "Modelica - A Unified Object-Oriented Language for System Modeling and Simulation." 12th European Conference on Object-Oriented Programming (ECOOP '98), Brussells, Belgium.

EIS Lab (1997). "EIS Lab TIGER Web." Georgia Institute of Technology Engineering Information Systems Laboratory. Internet WWW page, at URL: http://eislab.gatech.edu/tiger/ (version current at 12/29/1998).

EIS Lab (1998). "Product Data-Driven Analysis in a Missile Supply Chain." Georgia Institute of Technology Engineering Information Systems Laboratory. Internet WWW page, at URL: http;//eislab.gatech.edu/projects/proam (version current at December 31, 1998).

Gere, J. M. and S. P. Timoshenko (1990). Mechanics of Materials. PWS-KENT Publishing Company, Boston.

Gieck, Kurt, et al. (1990). Engineering Formulas. McGraw-Hill, West Germany.

Guyer, E. (1989). Handbook of Applied Thermal Design. McGraw-Hill, New York, NY.

Hardwick, M. (1994). "Towards Integrated Product Databases Using Views." Technical Report. Design and Manufacturing Institute, Rensselaer Polytechnic Institute, Troy, NY (Report No. 94003).

Helpenstein, H., T. Kenny, et al. (1997a). "Generic Engineering Analysis Model - Report on Standardisation Effort in GEM." Public Report. ESPRIT (Report No. EXPRIT CIME 8894 - Task 4400 - No. D4401).

Helpenstein, H., T. Kenny, et al. (1997b). "Generic Engineering Analysis Model - Improving Integration in Engineering Analysis." Public Report. ESPRIT (Report No. EXPRIT CIME 8894 - Task 4100 - No. D4101).

Hudson, S. E. (1998). "CUP User's Manual." Internet WWW page, at URL: www.cs.princeton.edu/~appel/modern/java/CUP/manual.html (version current at 08/12/1998).

Hunten, K. A. (1997). "CAD/FEA Integration with STEP AP 209 Technology and Implementation." Analysis Tools and Integration IPT Report. Lockheed Martin Tactical Aircraft Systems.

Institute for Interconnecting and Packaging Electronic Circuits (1999a). "Introduction to the GenCAM Standard." Internet WWW page, at URL: http://www.ipc.org/html/intro.html (version current at 01/18/1999).

Institute for Interconnecting and Packaging Electronic Circuits (1999b). "Welcome to IPC." Internet WWW page, at URL: www.ipc.org (version current at 01/18/1999).

ISO (1997). "Engineering Analysis Core Application reference model (EA C-ARM)." Proposal. International Organization for Standardization, Geneva, Switzerland.

ISO 10303-1 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 1: Overview and Fundamental Principles." International Organization for Standardization, Geneva, Switzerland.

ISO 10303-11 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 11: Description Methods: The EXPRESS Language Reference Manual." International Organization for Standardization.

ISO 10303-21 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure." International Organization for Standardization.

ISO 10303-42 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 42, Integrated generic resources: Geometric and topological representation." International Organization for Standardization.

ISO 10303-45 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 45 Integrated generic resources: Materials." International Organization for Standardization.

ISO 10303-104 (1995). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 104: Integrated application sources : Finite element analysis." International Organization for Standardization.

ISO 10303-105 (1997). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 105: Integrated application sources : Kinematics." International Organization for Standardization.

ISO 10303-203 (1994). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 203, Application Protocol: Configuration Controlled Design." International Organization for Standardization, Geneva, Switzerland.

ISO 10303-209 (1996). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 209, Application Protocol: Composite and metallic structural analysis and related design." International Organization for Standardization, Geneva, Switzerland.

ISO DIS 10303-210 (1993). "Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 210, Application Protocol: Printed Circuit Assembly Product Design Data." International Organization for Standardization, Geneva, Switzerland.

ISO TC184/SC4/WG5 N243 (1995). "EXPRESS-M Reference Manual." CiMiO Ltd., Surrey, England.

Kemper, A. H. and G. Moerkotte (1994). Object-oriented Database Management : Applications in Engineering and Computer Science. Prentice Hall, Englewood Cliffs, NJ.

Laurance, N. (1994). "A High-level View of STEP." Manufacturing Review 7(1): 39-46.

Leler, W. (1988). Constraint Programming Languages: Their Specification and Generation. Addison-Wesley Publishing Company.

Levine, J. R., T. Mason, et al. (1995). lex & yacc. O' Reilly and Associates, Inc.

Lewis, J. (1996). Manufacturing Engineer. Holaday Circuits Inc. Minnetonka, MN (personal communication).

Lipschutz, S. (1964). Set Theory and Related Topics. McGraw-Hill, New York, NY.

Marriott, K. and J. P. Stuckey (1998). Programming with Constraints, An Introduction. The MIT Press, Cambridge, Massachusetts.

Masquelier, V. (1996). "Java Beautifuller." Internet WWW page, at URL: http://www.santel.lu/SANTEL/contact/html/vm/java.html (version current at 04/11/99).

Mattsson, S. E. and H. Elmqvist (1998). "An Overview of the Modeling Language Modelica." Eurosim '98 Simulation Congress, Helsinki, Finland.

Mead, C. A. and L. A. Conway (1980). Introduction to VLSI Systems. Addison-Wesley Publishing Company, Reading, Massachusetts.

Morris, K. C., M. J. Mitchell, et al. (1992). "Database Management Systems in Engineering." National Institute of Standards and Technology, Gaithersburg, MD (Report No. NISTIR 4987).

National Institute of Standards and Technology (NIST) (1999). "EXPRESS Web Server." Internet WWW page, at URL: http://pitch.nist.gov/cgi-bin/sauder/express-server/server.cgi (version current at 04/08/99).

Owen, J. (1993). STEP, An Introduction. Information Geometers Ltd, Winchester, UK.

Peak, R., S. (1993). "Product Model-Based Analytical Models (PBAMs): A New Representation of Engineering Analysis Models." Doctoral Thesis, George W. Woodruff School of Mechanical Engineering. Georgia Institute of Technology, Atlanta, GA. 354 pages.

Peak, R. S. and R. E. Fulton (1993a). "Automating Routine Analysis in Electronic Packaging Using Product Model-Based Analytical Models (PBAMs), Part I: PBAM Overview." ASME Winter Annual Meeting 93-WA/EEP-23.

Peak, R. S. and R. E. Fulton (1993b). "Automating Routine Analysis in Electronic Packaging Using Product Model-Based Analytical Models (PBAMs), Part II: Solder Joint Fatigue Case Studies." ASME Winter Annual Meeting 93-WA/EEP-24.

Peak, R. S. and R. E. Fulton (1993c). "A Multi-Representation Approach to CAD/CAE Integration: Research Overview." Interim Report. Manufacturing Research Center, Georgia Institute of Technology, Atlanta, GA (Report No. MS-93-03).

Peak, R. S., A. Scholand, et al. (1996). "On the Routinization of Analysis for Physical Design." 1996 ASME International Mechanical Engineering Congress and Exposition, Application of CAE/CAD to Electronic Systems EEP-Vol.18.

Peak, R. S., R. E. Fulton, et al. (1997). "Thermomechanical CAD/CAE Integration in the TIGER PWA Toolset." InterPACK '97, Advances in Electronic Packaging EEP-Vol. 19-1: 957-962.

Peak, R. S., R. E. Fulton, et al. (1998). "Integrating Engineering Design and Analysis Using a Multi-Representation Approach." Engineering with Computers 14: 93-114.

Peak, R. S., R. E. Fulton, et al. (1999). "Phase I Report: Pilot Demonstration of STEP-based Stress Templates." GIT Project Report. CALS Technology Center's Engineering Information Systems Laboratory, Atlanta, GA (Report No. E-15-647, The Boeing Company Contract W309702).

Prather, M. H. and R. A. Amador (1997). Product Simulation Integration for Structures. MacNeal-Schendler Corporation Aerospace Users Conference, New Port Beach, CA, The Boeing Commercial Airplane Group (BCAG).

Rosen, D. W. (1992). "A Feature-Based Representation to Support the Design Process and the Manufacturability Evaluation of Mechanical Components." PhD Dissertation, Department of Mechanical Engineering. University of Massachusetts. 200 pages.

Rosen, D. W., J. R. Dixon, et al. (1991). "A Methodology for Conversions of Feature-based Representations." Proceedings of the Design Theory and Methodology - DTM '91 31: 45-51.

Rosen, D. W., J. R. Dixon, et al. (1994). "Conversions of Feature-Based Design Representations Using Grammar Parsing." Journal of Mechanical Design 116: 785-792.

Rosen, K. H. (1995). Discrete Mathematics and its Applications. McGraw-Hill, New York, NY.

Rumbaugh, J., M. Blaha, et al. (1991). Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ.

Sahu, K. and I. R. Grosse (1994). "Concurrent Iterative Design and the Integration of Finite Element Analysis Results." Engineering with Computers 10(4): 245-257.

Sargent, R. G., J. D. Tew, et al. (1994). "Verification and validation of simulation models." 1994 Winter Simulation Conference, Lake Buena Vista, FL, IEEE.

Sauter, G. and W. Käfer (1996). "BRIITY - A Mapping Language Bridging Heterogeneity." ITG/GI/GMA Conf. "Software Technology in Automation and Communication", München, Germany.

Schenck, D. and P. Wilson (1994). Information Modeling the EXPRESS Way. Oxford University Press, New York, NY.

Scholand, A. J., R. S. Peak, et al. (1997). "The Engineering Service Bureau - Empowering SMEs to Improve Collaboratively Developed Products." CALS Expo USA Track 2, Session 4.

SCRA (1997). "Team Integrated Electronic Response." Internet WWW page, at URL: http://www.scra.org/tiger/tiger.html (version current at 12/29/1998).

Shephard, M. S. and R. Wentorf (1994). "Toward the Implementation of Automated Analysis Idealization Control." Applied Numerical Mathematics 14(1-3): 105-124.

Shephard, M. S., E. V. Korngold, et al. (1990). Design Systems Supporting Engineering Idealizations. Geometric Modeling for Product Engineering. M. J. Wozny, J. U. Turner and K. Preiss. Elsevier Science Publishers, North-Holland: 279-299.

Shephard, M. S., P. L. Baehmann, et al. (1990). "Framework for the Reliable Generation and Control of Analysis Idealizations." Computer Methods in Applied Mechanics and Engineering 82: 257-280.

Shephard, M. S., P. L. Baehmann, et al. (1992). "Methodology for the Integration of Global/Local Thermal and Thermo-Mechanical Analyses of Multichip Modules." Computer Aided Design in Electronic Packaging - ASME EEP- Vol 3: 65-72.

Shephard, M. S., T. L. Sham, et al. "Global/Local Heat Conduction and Thermomechanical Analyses of Multichip Modules." : 123-134.

Shigley, J. E. and C. R. Mischke (1989). Mechanical Engineering Design. McGraw-Hill, New York, NY.

Si Alhir, S. (1998). UML in a Nutshell. O' Reilly and Associates, Inc., Sebastopol, CA.

Spooner, D. (1993). "Creating a Product Database with ROSE." Technical Report. Design and Manufacturing Institute, Rensselaer Polytechnic Institute, Troy, NY (Report No. 93054).

Spooner, D. and M. Hardwick (1993). Using Persistent Object Technology to Support Concurrent Engineering Systems. Concurrent Engineering: Methodology and Applications. P. Gu and A. Kusiak. Elsevier Science, New York**:** 205-234.

Spooner, D., M. Hardwick, et al. (1995). "The EXPRESS-V Language Manual." Internet WWW page, at URL: http://www.rdrc.rpi.edu/EXPRESS-V/homepage.html (version current at June, 29 1997).

Spooner, D., M. Hardwick, et al. (1996). "The EXPRESS-X Language Manual." Internet WWW page, at URL: http://www.rdrc.rpi.edu/EXPRESS-X/homepage.html (version current at June 29, 1997).

STEP Tools Inc (1997a). Java Tools Reference Manual. Troy, NY, STEP Tools Inc,.

STEP Tools Inc (1997b). ROSE Library Reference Manual. Troy, NY, STEP Tools Inc.

STEP Tools Inc (1997c). ST-Developer Reference Manual (Version 1.6). Troy, NY, STEP Tools Inc,.

Stiteler, M. R. (1996). "An automated system for measuring PWB/PWBA warpage during simulation of the infrared reflow and wave soldering processes, and during operational thermal cycling." Masters Thesis, School of Mechanical Engineering. Georgia Institute of Technology, Atlanta. 119 pages.

Sun Microsystems (1998). "Java Technology Home Page." Internet WWW page, at URL: www.javasoft.com (version current at 08/16/98).

Tamburini, D. R., R. S. Peak, et al. (1996). "Populating Product Data for Engineering Analysis With Applications to Printed Wiring Assemblies." 1996 ASME International Mechanical Engineering Congress and Exposition EEP-18: 33-46.

Tamburini, D. R., R. S. Peak, et al. (1997). "Driving PWA Thermomechanical Analysis from STEP AP210 Product Models." 1997 ASME International Mechanical Engineering Congress and Exposition, CAE/CAD and Thermal Management Issues in Electronic Systems EEP-Vol. 23/HTD-Vol. 356: 33-45.

Tamburini, D. R. (1999). "APM Protocol Prototype Implementation." Internet WWW page, at URL: http://eislab.gatech.edu/people/tamburini/thesis/code (version current at 01/23/1999).

Thomas, C. H., C. E. Leiserson, et al. (1992). Introduction to Algorithms, McGraw-Hill, New York, NY.

Wentorf, R. and M. S. Shephard (1993). "User Interface Functions for an Analysis Idealization System." Microcomputers in Civil Engineering 8(2): 85-95.

Wilson, M. (expected 1999). "The Constrained Object (COB) Representation for Engineering Analysis Integration." Masters Thesis, George W. Woodruff School of Mechanical Engineering. Georgia Institute of Technology, Atlanta, GA.

Wilson, M., R. S. Peak, et al. (1999). "XaiTools Users Guide." GIT Project Report. CALS Technology Center's Engineering Information Systems Laboratory, Atlanta, GA (Report No. E-15-647 - Attachment)

Wilson, P. (1991). "Modeling Languages Compared." Rensselear Design Research Center. Rensselear Polytechnic Institute, Troy, NY.

Wilson, P. R. (1996). "EXPRESS and Set Theory (DRAFT)." Internet WWW page, at URL: http://www.nist.gov/sc4/wg_qc/wg11/express/tecpaper/sets.ps (version current at June 29, 1997).

Wilson, P. (1998). "EXPRESS Tools and Services." ISO EXPRESS Committee, Seattle.

Wolfram, S. (1996). The Mathematica Book. Wolfram Media/ Cambridge University Press.

Yeh, C.-P. (1992). "An Integrated Information Framework for Multidisciplinary PWB Design." Doctoral Thesis, School of Mechanical Engineering. Georgia Institute of Technology, Atlanta, GA. pp 462-474

APPENDICES

APPENDIX A

STEP (ISO 10303) OVERVIEW[70]

STEP (Standard for the Exchange of Product Data) is an international standard for the exchange of product data between engineering systems. STEP is defined in (ISO 10303-203 1994) as follows:

> *"ISO 1030 is an international standard for the computer-sensible representation and exchange of product data. The objective is to provide a mechanism capable of describing product data throughout the lifecycle of a product, independent of any particular system. The nature of this description makes it suitable not only for file exchange, but also as a basis for implementing and sharing product databases and archiving."*

Development of STEP started in the early 1980s as a response to the deficiencies of the existing generation of product data standards and specifications and to stop the proliferation of interim or alternative solutions to these problems. Since its Initial Release in March of 1994, new STEP parts and numerous application protocols are being developed, different organizations and consortia are implementing prototype applications of the standard, and several vendors are developing commercial translators. STEP's ultimate intent is to become a single and better standard that supports the information needs of all aspects of product life cycle. The international STEP development effort is organized by ISO's Technical Committee TC184 ("Industrial Automation Systems and Integration"), Sub-committee SC4 ("Industrial Data"). In the United States, the effort is coordinated by USPRO (formerly IPO, the IGES/PDES Organization).

STEP is not one, but a family of standards divided into two main groups: the STEP data models and the tools to create these models. The STEP data models can cover any type of

[70] Unless otherwise indicated, the material in this section has been extracted from (Al-Timimi and MacKrell 1996; Hardwick 1994; ISO 10303-1 1994; Laurance 1994; Owen 1993).

product data, not just geometry. So instead of one data model to represent geometry to replace IGES or other existing data exchange standards, STEP provides the possibility of creating an unlimited number of data models. Thus, different models, called ***Application Protocols (APs)***, can be used to specify the schemas for a product or products in different application areas. Application Protocols (APs) are information models that specify the structure of the data for the exchange of information between applications of a specific domain. An Application Protocol contains definitions of the application domain and test procedures to validate that a given implementation conforms to the Application Protocol. It is expected that several hundred AP's will be developed to support the many industrial applications that STEP is expected to serve. Examples of application protocols are:

- 10303-202: Associative Drafting;
- 10303-203: Configuration Controlled Design;
- 10303-210: Printed Circuit Assembly Product Design Data;
- 10303-214: Core Data for Automotive Mechanical Design Processes;
- 10303-218: Ship Structures;
- 10303-221: Functional Data and Schematic Representation for Process Plans.

This thesis mentions two Applications Protocols: AP210 (ISO 10303-11 1994; Schenck and Wilson 1994), which describes the structure of the data needed to provide a manufacturable description of a PCA, and AP203 (ISO 10303-21 1994) which specifies the structures for the exchange between application systems of configuration controlled three-dimensional product definition data of mechanical parts and assemblies.

Some parts of the Applications Protocols are common to many application areas. To support this, STEP provides two other types of data models: Integrated Resources and Application Interpreted Constructs. ***Integrated Resources (IRs)*** are STEP data models that can be used in more that one application area. Unlike APs, they do not have an application context. They represent generic or semi-generic primitive data models that can be incorporated in other more complex data models, such as Applications Protocols or Application Interpreted Constructs. There are, in turn, two types of Integrated Resources: Generic Integrated Resources ("40" series) and Application Integrated Resources ("100"

series). ***Generic Integrated Resources*** are entirely independent of specific application areas. They represent the model data entities that may be used to support many different disciplines. Examples are:

- 10303-42: Geometric and Topological Representation;
- 10303-44: Product Structure Configuration;
- 10303-45: Integrated Generic Resources: Materials;
- 10303-47: Integrated Generic Resources: Shape Tolerances.

Application Integrated Resources, on the other hand, define data entities that are common for a given application area. They help define data that can be used in a number of closely related applications. Examples are:

- 10303-101: Drafting Resources;
- 10303-103: Integrated Application Resources: Electrical/Electronics Connectivity;
- 10303-104: Integrated Application Resources: Finite Element Analysis;
- 10303-105: Integrated Application Resources: Kinematics;

Application Interpreted Constructs (AICs) are also STEP data models. Like Application Protocols, they contain descriptions of data within an application context. However, they represent core specification that can be included in more than one Application Protocol. Therefore, together with Integrated Resources, Application Interpreted Constructs represent the building blocks for Application Protocols thus enabling the re-usability of generic and semi-generic data models. If Applications Protocols are viewed as *products* and Integrated Resources as *components*, then Application Interpreted Constructs may be viewed as *assemblies*. Examples of Application Interpreted Constructs currently under development by ISO include:

- 10303-501: Edge-based Wireframe;
- 10303-505: Drawing Structure;
- 10303-508: Non-manifold Surface;

- 10303-509: Manifold Surface;
- 10303-512: Faceted B-rep;
- 10303-515: Constructive Solid Geometry.

The STEP tools used to create data models are also divided into a number of sub-groups:

- ***Description Methods:*** the formal languages used in describing STEP data models. These include EXPRESS (introduced below) and EXPRESS-I[71].
- ***Implementation Methods:*** these include the STEP Exchange (Physical) File format (introduced below) and SDAI, the STEP Application Programming Interface[71].
- ***Conformance Testing Methodology:*** used to verify that a given STEP data model implementation conforms to its specification[71].

Each of the Application Protocols, Integrated Resources and Application Interpreted Constructs are specified using the ***EXPRESS*** language (Spooner, Hardwick et al. 1995). EXPRESS is a formal textual data definition language that provides the mechanism for the description of product data. It permits the definition of resource constructs from data elements, constraints, relationships, rules and functions. EXPRESS has a companion graphical form called ***EXPRESS-G***.

The STEP standard also specifies the physical format of the exchange file (Spooner, Hardwick et al. 1996). Exchange files that conform to this format are called ***STEP Exchange Files***, ***STEP Files*** or, more commonly, ***Part 21 Files*** (from the STEP part in which they are defined). Two applications may exchange a Part 21 file as long as they are both aware of the EXPRESS schema that was used to generate the data.

A topic within STEP that is of special interest for this thesis is the topic of ***STEP mapping languages***. STEP mapping languages resulted from the need to be able to easily create views of STEP product models tailored to individual application systems. STEP product models must be complete and unambiguous, and must support the information requirements of a range of application systems in a given domain. As a result, they are large

[71] Not discussed in this appendix.

and contain details that many individual application systems may not need (Spooner, Hardwick et al. 1995). These views omit unnecessary details and are conceptually easier to understand than the STEP representations from which they are derived.

STEP mapping languages allow the definition of views of a product model defined in EXPRESS. They are usually extensions of EXPRESS that provide the capability to describe how data that conforms to one or more source schemas is mapped into a target schema. The STEP community is currently developing ***EXPRESS-X*** (Spooner, Hardwick et al. 1996), a combination of earlier STEP mapping languages such as EXPRESSS-V (Spooner, Hardwick et al. 1995), EXPRESS-M (ISO TC184/SC4/WG5 N243 1995) and BRIITY (Sauter and Käfer 1996). EXPRESS-X is likely to become the standard STEP mapping language in the near future.

Defining a mapping requires the definition of three schemas, two of which are ordinary EXPRESS schemas. The first of these schemas is called the *base schema* and defines the schema for the original product model from which the view will be derived. The second schema is the *view* or *target schema*, which defines the product model for the materialized view – that is, the entity types that will be in the view and the attributes for each of these entity types. There could be more than one source or target schema. The third schema is the mapping schema, which takes advantage of the extensions to EXPRESS that are in the mapping language. The mapping schema defines the mappings between entities in the base schema and the view schema. Implementations of these mapping languages normally include a compiler for validating the syntax of the definition of the views and a run-time system for materializing them (create the instances in the target schema).

The complexity of the mappings is determined by the amount of semantic mismatch between the source and the target schemas. Mappings between two versions of the same schema, for example, have small semantic mismatch and are relatively easy to describe. On the other end, mappings between two schemas that describe a product from the point of view of two different engineering disciplines (e.g., electrical and mechanical), will have a large semantic mismatch. These more complex mappings are more representative of the types of mappings encountered in this research.

STEP and its textual conceptual schema language EXPRESS were selected for this research because they provide the neutral mechanisms for describing and exchanging product

information and because there are several development tools commercially available (Spooner 1993; STEP Tools Inc 1997a; STEP Tools Inc 1997b; STEP Tools Inc 1997c) to aide in the development of STEP applications. Another reason for selecting STEP is its growing international acceptance by industry, government and academia.

APPENDIX B

APM RELATIONSHIPS REFERENCE

<*attribute*> ∈* <*object domain*> (Definition 38-29)

<*attribute*> "directly belongs to" <*object domain*> if <*attribute*> is an attribute (local or inherited) of <*object domain*>.

When this is true, the container domain of <*attribute*> is equal to <*object domain*>.

<*attribute*> ∈* <*multi-level domain*> (Definition 38-30)

<*attribute*> "directly belongs to" <*multi-level domain*> if <*attribute*> is a level of <*multi-level domain*>.

When this is true, the container domain of <*attribute*> is equal to <*multi-level domain*>.

<*attribute*> ∈˜ <*complex domain*>(Definition 38-31)

<*attribute*> "indirectly belongs" to <*complex domain*> if it directly or indirectly belongs to any of the attributes of <*complex domain*>.

Direct belonging implies indirect belonging. In other words:

<*attribute*> ∈* <*complex domain*> ⌟ <*attribute*> ∈˜ <*complex domain*>.

<*attribute*$_1$> ∈* <*attribute*$_2$> (Definition 38-32)

<*attribute*$_1$> "directly belongs" to <*attribute*$_2$> if <*attribute*$_1$> directly belongs to the domain of <*attribute*$_2$>.

<*attribute*$_1$> ∈˜ <*attribute*$_2$> (Definition 38-33)

<*attribute*$_1$> "indirectly belongs" to <*attribute*$_2$> if <*attribute*$_1$> indirectly belongs to the domain of <*attribute*$_2$>.

<*instance*> ∈^{i} <*domain*> (Definition 38-35)

<*instance*> "is an instance of" <*domain*> if the domain of <*instance*> is equal to <*domain*>.

<*domain*> ∈* <*domain set*> (Definition 38-56)

<*domain*> "belongs to" <*domain set*> if <*domain*> is one of the member domains of <*domain set*>.

<*source set*> ∈* <*apm*> (Definition 38-74)

<*source set*> "belongs to" <*apm*> if <*source set*> belongs to the list of source sets of <*apm*>.

<*source set link*> ∈* <*apm*> (Definition 38-75)

<*source set link*> "belongs to" <*apm*> if <*source set link*> belongs to the list of source sets links of <*apm*>.

APPENDIX C

APM DEFINITION LANGUAGES

C.1 APM Structure Definition Language

Generic APM Structure Definition Language Grammar

Terminals (keywords that should be recognized as tokens by the scanner)

APM
END_APM
SOURCE_SET
END_SOURCE_SET
ROOT_DOMAIN
DOMAIN
END_DOMAIN
MULTI_LEVEL_DOMAIN
END_MULTI_LEVEL_DOMAIN
SUBTYPE_OF
LIST
OF
MULTI_LEVEL
IDEALIZED
ESSENTIAL
PRODUCT_RELATIONS
PRODUCT_IDEALIZATION_RELATIONS
REAL
STRING
LINK_DEFINITIONS
END_LINK_DEFINITIONS

Non-Terminals (intermediate variables created in the grammar actions)

APM apm_definition
source_set_definitions
APMSourceSet source_set_definition
domains
APMDomain domain
attributes
APMAttribute attribute
relations
productRelations
productIdealizationRelations
listOfProductIdealizationRelations
listOfProductRelations
APMProductRelation productRelation
APMProductIdealizationRelation productIdealizationRelation
link_definitions
link_defs
APMSourceSetLink link_def

Global variables used by the grammar actions

APMPrimitiveDomain realDomain
APMPrimitiveDomain stringDomain
APMPrimitiveAggregateDomain listOfReals
APMPrimitiveAggregateDomain listOfStrings
ListOfAPMAttributes tempListOfAPMAttributes
ListOfAPMDomains tempListOfAPMDomains
ListOfAPMSourceSets tempListOfAPMSourceSets
ListOfAPMRelations tempListOfAPMRelations
Dictonary tableOfDefinedAPMDomains
ListOfAPMSourceSetLinks tempListOfAPMSourceSetLinks

Grammars

APM

apm_definition ·

APM <apm name> ; source_set_definitions link_definitions END_APM ;
{
- Create a new APM instance and return it to the operation calling the parser:
  RESULT = new APM(<apm name> , tempListOfAPMSourceSets , tempListOfAPMSourceSetLinks)
}

OR

APM <apm name> ; source_set_definitions END_APM ;
{
- Create a new APM instance and return it to the operation calling the parser:
  RESULT = new APM(<apm name> , tempListOfAPMSourceSets)
}

SOURCE SETS

source_set_definitions ·

source_set_definition

OR

source_set_definitions source_set_definition

source_set_definition ·

SOURCE_SET <source set name> ROOT_DOMAIN <root domain name> ; domains END_SOURCE_SET ;
{

- Local variables:
  APMComplexDomain rootDomain

- If the real domain as been created (i.e., if realDomain is not null):
  - Add it to the list of defined domains of this source set:
    tempListOfAPMDomains.addElement(realDomain)

  - Add it to the table of defined domains of this source set:
    tableOfDefinedAPMDomains.put("REAL" , realDomain)

- If the string domain as been created (i.e., if stringDomain is not null):
  - Add it to the list of defined domains of this source set:
    tempListOfAPMDomains.addElement(stringDomain)

  - Add it to the table of defined domains of this source set:
    tableOfDefinedAPMDomains.put("STRING" , stringDomain)

- If the listOfReals domain as been created (i.e., if listOfReals is not null):
  - Add it to the list of defined domains of this source set:
    tempListOfAPMDomains.addElement(listOfReals)

  - Add it to the table of defined domains of this source set:
    tableOfDefinedAPMDomains.put("ListOfReals" , listOfReals)

- If the listOfStrings domain as been created (i.e., if listOfStrings is not null):
  - Add it to the list of defined domains of this source set:
    tempListOfAPMDomains.addElement(listOfStrings)

- Add it to the table of defined domains of this source set:
  tableOfDefinedAPMDomains.put("ListOfStrings" , listOfStrings)

- Clear the primitive domains for the next source set:
  realDomain = null
  stringDomain = null
  listOfReals = null
  listOfStrings = null

- Check that all domains referenced by the domains in this source set have been defined in this source set.
  checkDependencies( )

- Get the root domain from the tempListOfAPMDomains of this source set:
  rootDomain = getDomainFromList(<root domain name> , tempListOfAPMDomains)

- Create the source set:
  RESULT = new APMSourceSet(<source set name> , tempListOfAPMDomains , rootDomain)

- Add the source set to the list of source sets of the apm:
  tempListOfAPMSourceSets.addElement(RESULT)

- Assign RESULT to the sourceSet attribute of each domain in tempListOfAPMDomains:
  setSourceSet(RESULT , tempListOfAPMDomains)

- Clear tempListOfAPMDomains and tableOfDefinedAPMDomains for the next source set.

}

DOMAINS

domains ·

domain

OR

domains domain

domain ·
DOMAIN <domain name> ; attributes END_DOMAIN ;
{

- Local variables:
  APMSourceSet dummySourceSet

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , tempListOfAPMAttributes , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:
  setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes for the next domain.

}

OR

DOMAIN <domain name> SUBTYPE_OF <supertype domain name>; attributes END_DOMAIN;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummySupertypeDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy supertype domain:
  dummySupertypeDomain = new APMObjectDomain(<supertype domain name> , dummySourceSet)

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , dummySupertypeDomain ,
  tempListOfAPMAttributes , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:
  setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes for the next domain.
}

OR

DOMAIN <domain name> ; attributes relations END_DOMAIN;
{
- Local variables:
  APMSourceSet dummySourceSet

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , tempListOfAPMAttributes ,
  tempListOfAPMRelations , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:
  setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes and tempListOfAPMRelations for the next domain.
}

OR

DOMAIN <domain name> SUBTYPE_OF <supertype domain name> ; attributes relations END_DOMAIN;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummySupertypeDomain

- Create a dummy source set:

dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy supertype domain:
  dummySupertypeDomain = new APMObjectDomain(<supertype domain name> , dummySourceSet)

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , dummySupertypeDomain ,
  tempListOfAPMAttributes , tempListOfAPMRelations , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:
  setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes and tempListOfAPMRelations for the next domain.

}

OR

DOMAIN <domain name> SUBTYPE_OF <supertype domain name>; relations END_DOMAIN;
{

- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummySupertypeDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy supertype domain
  dummySupertypeDomain = new APMObjectDomain(<supertype domain name> , dummySourceSet)

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , dummySupertypeDomain ,
  tempListOfAPMRelations , dummySourceSet)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMRelations for the next domain.

}

OR

DOMAIN <domain name> ; END_DOMAIN;
{

- Local variables:
  APMSourceSet dummySourceSet

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , dummySourceSet)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

}

OR

DOMAIN <domain name> SUBTYPE_OF <supertype domain name> ; END_DOMAIN ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummySupertypeDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy supertype domain:
  dummySupertypeDomain = new APMObjectDomain(<supertype domain name> , dummySourceSet)

- Create a new APMObjectDomain:
  RESULT = new APMObjectDomain(<domain name> , dummySupertypeDomain , dummySourceSet)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)
}

OR

MULTI_LEVEL_DOMAIN <domain name> ; attributes END_MULTI_LEVEL_DOMAIN;
{
- Local variables:
  APMSourceSet dummySourceSet

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a new APMMultiLevelDomain:
  RESULT = new APMMultiLevelDomain(<domain name> , tempListOfAPMAttributes , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:
  setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes for the next domain.
}

OR

MULTI_LEVEL_DOMAIN <domain name> ; attributes relations END_MULTI_LEVEL_DOMAIN;
{
- Local variables:
  APMSourceSet dummySourceSet

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a new APMMultiLevelDomain:
  RESULT = new APMMultiLevelDomain (<domain name> , tempListOfAPMAttributes ,
  tempListOfAPMRelations , dummySourceSet)

- Assign RESULT to the variable containerDomain of each attribute in tempListOfAPMAttributes:

setContainerDomain(tempListOfAPMAttributes , RESULT)

- Add RESULT to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(<domain name> , RESULT)

- Add RESULT to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(RESULT)

- Clear tempListOfAPMAttributes and tempListOfAPMRelations for the next domain.

}

ATTRIBUTES

attributes ·
attribute

OR

attributes attribute

attribute ·
<attribute_name> ; REAL ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the real domain (if it hasn't been created already):
  realDomain = new APMPrimitiveDomain("REAL" , dummySourceSet)

- Create a new APMPrimitiveAttribute:
  RESULT = new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain ,
  APMPrimitiveAttribute.PRODUCT , realDomain)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)
}

OR

<attribute_name> ; STRING ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the string domain (if it hasn't been created already):
  stringDomain = new APMPrimitiveDomain("STRING" , dummySourceSet)

- Create a new APMPrimitiveAttribute:

RESULT = new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain , APMPrimitiveAttribute.PRODUCT , stringDomain)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)

}

OR

ESSENTIAL <attribute_name> ; REAL ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the real domain (if it hasn't been created already):
  realDomain = new APMPrimitiveDomain("REAL" , dummySourceSet)

- Create a new APMPrimitiveAttribute:
  RESULT = new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain , APMPrimitiveAttribute.ESSENTIAL , realDomain)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)

}

OR

ESSENTIAL <attribute_name> ; STRING ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the string domain (if it hasn't been created already):
  stringDomain = new APMPrimitiveDomain("STRING" , dummySourceSet)

- Create a new APMPrimitiveAttribute:
  RESULT = new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain , APMPrimitiveAttribute.ESSENTIAL , stringDomain)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)

}

OR

IDEALIZED <attribute_name> ; REAL ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:

```
        dummySourceSet = new APMSourceSet( "dummySourceSet" )

    ▸ Create a dummy container domain:
        dummyContainerDomain = new APMObjectDomain( "Dummy Name" , dummySourceSet )

    ▸ Create the real domain (if it hasn't been created already):
        realDomain = new APMPrimitiveDomain( "REAL" , dummySourceSet )

    ▸ Create a new APMPrimitiveAttribute:
        RESULT =  new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain ,
        APMPrimitiveAttribute.IDEALIZED , realDomain )

    ▸ Add RESULT to tempListOfAPMAttributes:
        tempListOfAPMAttributes.addElement( RESULT )
}

OR

IDEALIZED <attribute_name> ; STRING ;
{
    ▸ Local variables:
        APMSourceSet dummySourceSet
        APMObjectDomain dummyContainerDomain

    ▸ Create a dummy source set:
        dummySourceSet = new APMSourceSet( "dummySourceSet" )

    ▸ Create a dummy container domain:
        dummyContainerDomain = new APMObjectDomain( "Dummy Name" , dummySourceSet )

    ▸ Create the string domain (if it hasn't been created already):
        stringDomain = new APMPrimitiveDomain( "STRING" , dummySourceSet )

    ▸ Create a new APMPrimitiveAttribute:
        RESULT =  new APMPrimitiveAttribute(<attribute_name> , dummyContainerDomain ,
        APMPrimitiveAttribute.IDEALIZED , stringDomain )

    ▸ Add RESULT to tempListOfAPMAttributes:
        tempListOfAPMAttributes.addElement( RESULT )
}

OR

<attribute_name> ; <domain name> ;
{
    ▸ Local variables:
        APMSourceSet dummySourceSet
        APMObjectDomain dummyContainerDomain
        APMObjectDomain dummyDomain

    ▸ Create a dummy source set:
        dummySourceSet = new APMSourceSet( "dummySourceSet" )

    ▸ Create a dummy container domain:
        dummyContainerDomain = new APMObjectDomain( "Dummy Name" , dummySourceSet )

    ▸ Create a dummyDomain:
        dummyDomain = new APMObjectDomain(<domain name> , dummySourceSet )

    ▸ Create a new APMObjectAttribute:
        RESULT =  new APMObjectAttribute( <attribute_name> , dummyContainerDomain , dummyDomain )

    ▸ Add RESULT to tempListOfAPMAttributes:
        tempListOfAPMAttributes.addElement( RESULT )
}
```

OR

<attribute_name> ; MULTI_LEVEL <domain name> ;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain
  APMMultiLevelDomain dummyDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create a dummyDomain:
  dummyDomain = new APMMultiLevelDomain(<domain name> , dummySourceSet)

- Create a new APMMultiLevelAttribute:
  RESULT = new APMMultiLevelAttribute(<attribute_name> , dummyContainerDomain , dummyDomain)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)
}

OR

<attribute_name> ; LIST [<low bound> , <high_bound>] OF <domain name>;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyDomainOfElements
  String newDomainName
  APMComplexAggregateDomain attributeDomain
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy domain for the elements of the aggregate:
  dummyDomainOfElements = new APMObjectDomain(<domain name> , dummySourceSet)

- Create a name consisting of <domain name> prefixed with "ListOf" and suffixed with "s":
  newDomainName = "ListOf" + <domain name> + "s"

- Create the domain of the attribute:
  attributeDomain = new APMComplexAggregateDomain(newDomainName , dummySourceSet ,
  dummyDomainOfElements)

- Add attributeDomain to tempListOfAPMDomains:
  tempListOfAPMDomains.addElement(newAggregateDomain)

- Add attributeDomain to tableOfDefinedAPMDomains:
  tableOfDefinedAPMDomains.put(newDomainName , newAggregateDomain)

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create a new APMComplexAggregateAttribute:
  RESULT = new APMComplexAggregateAttribute(<attribute_name> , dummyContainerDomain ,
  newAggregateDomain , <low bound> , <high bound>)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)
}

OR

<attribute_name> ; LIST [<low bound> , <high bound>] OF REAL;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the real domain (if it hasn't been created already):
  realDomain = new APMPrimitiveDomain("REAL" , dummySourceSet)

- Create the domain listOfReals (if it hasn't been created already):
  listOfReals = new APMPrimitiveAggregateDomain("ListOfReals" , dummySourceSet , realDomain)

- Create a new APMPrimitiveAggregateAttribute:
  RESULT = new APMPrimitiveAggregateAttribute (<attribute_name> , dummyContainerDomain , listOfReals , <low bound> , <high bound>)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)

}

OR

<attribute_name> ; LIST [BOUND:b1 , BOUND:b2] OF STRING;
{
- Local variables:
  APMSourceSet dummySourceSet
  APMObjectDomain dummyContainerDomain

- Create a dummy source set:
  dummySourceSet = new APMSourceSet("dummySourceSet")

- Create a dummy container domain:
  dummyContainerDomain = new APMObjectDomain("Dummy Name" , dummySourceSet)

- Create the string domain (if it hasn't been created already):
  stringDomain = new APMPrimitiveDomain("STRING" , dummySourceSet)

- Create the domain listOfStrings (if it hasn't been created already):
  listOfStrings = new APMPrimitiveAggregateDomain(" ListOfStrings " , dummySourceSet , stringDomain)

- Create a new APMPrimitiveAggregateAttribute:
  RESULT = new APMPrimitiveAggregateAttribute (<attribute_name> , dummyContainerDomain , listOfStrings , <low bound> , <high bound>)

- Add RESULT to tempListOfAPMAttributes:
  tempListOfAPMAttributes.addElement(RESULT)

}

RELATIONS

relations ·

productIdealizationRelations productRelations

OR

productRelations productIdealizationRelations

OR

productIdealizationRelations

OR

productRelations

productIdealizationRelations · PRODUCT_IDEALIZATION_RELATIONS listOfProductIdealizationRelations

listOfProductIdealizationRelations ·
productIdealizationRelation

OR

listOfProductIdealizationRelations productIdealizationRelation

productIdealizationRelation ·

<relation name>; "<relation expression>" ;
{
- Create a new APMProductIdealizationRelation:
  RESULT = new APMProductIdealizationRelation(<relation name> , <relation expression>)

- Add RESULT to tempListOfAPMRelations:
  tempListOfAPMRelations.addElement(RESULT)
}

productRelations · PRODUCT_RELATIONS listOfProductRelations

listOfProductRelations ·
productRelation

OR

listOfProductRelations productRelation

productRelation ·
<relation name>; "<relation expression>" ;
{
- Create a new APMProductRelation:
  RESULT = new APMProductRelation (<relation name> , <relation expression>)

- Add RESULT to tempListOfAPMRelations:
  tempListOfAPMRelations.addElement(RESULT)
}

SOURCE SET LINKS

link_definitions · LINK_DEFINITIONS link_defs END_LINK_DEFINITIONS;
{
- Do nothing.
}

link_defs ·
link_def

OR

link_defs link_def

link_def ·
<full attribute name 1> <logical operator> <full attribute name 2> ;
{
- Local variables:
  ListOfStrings fullAttributeName1
  ListOfStrings fullAttributeName2
  APMSourceSetLinkAttribute key1
  APMSourceSetLinkAttribute key2

- Create a list of strings with each dot-separated name in <full attribute name 1>:
  fullAttributeName1 = stripFullAttributeName(<full attribute name 1>)

- Create a list of strings with each dot-separated name in <full attribute name 2>:
  fullAttributeName2 = stripFullAttributeName(<full attribute name 2>)

- Create a new APMSourceSetLinkAttribute with fullAttributeName1:
  key1 = new APMSourceSetLinkAttribute(fullAttributeName1)

- Create a new APMSourceSetLinkAttribute with fullAttributeName2:
  key2 = new APMSourceSetLinkAttribute(fullAttributeName2)

- Create a new APMSourceSetLink:
  RESULT = new APMSourceSetLink(key1 , key2 , <logical operator>)

- Add RESULT to tempListOfAPMSourceSetLinks:
  tempListOfAPMSourceSetLinks.addElement(RESULT)
}

UTILITY OPERATIONS USED IN THE GRAMMAR ACTIONS

- void checkDependencies()
  - Checks each domain in table tableOfDefinedAPMDomains to make that all the domains used to define it are defined in the same source set. A domain may reference to other domains in the definition of its attributes, or when it declares another domain as its supertype.
  - Redirects any pointers to dummy source sets defined in the previous grammar actions to the right source set (the one being created when this operation is called).
  - Redirects any pointers to dummy domains used in the previous grammar actions to the actual domains stored in tableOfDefinedAPMDomains.

- void list.addElement(object o)
  - Adds object o to the end of list.

- void table.add(String key, object o)
  - Adds key-value pair (key , o) to table.

- APMDomain getDomainFromList(String domainName, ListOfAPMDomains listOfAPMDomains)
  - Gets the APMDomain in listOfAPMDomains whose domainName is domainName.

- setSourceSet(APMSourceSet set , ListOfAPMDomains listOfAPMDomains)
  - Sets the value of variable sourceSet of each APMDomain in listOfAPMDomains to set.

- void setContainerDomain(ListOfAPMAttributes listOfAPMAttributes , APMComplexDomain dom)
  - Sets the value of variable containerDomain of each APMAttribute in listOfAPMAttributes to dom.

- ListOfStrings stripFullAttributeName(String fullAttributeName)
  - Takes fullAttributeName (which is normally a string of several names separated by dots) and creates a list of strings with each of the dot-separated names in fullAttributeName.

Jlex APM Structure Definition Language Lexer Specification

```
package apm.parser;
import java.io.*;
import java_cup.runtime.Symbol;
import apm.parser.APMParserSym;


%%

%{
  static APMLexer foo;
  static BufferedReader fileToScan;

  public static void init( String fileName )
  {


    try {
       fileToScan = new BufferedReader( new FileReader( fileName ) );
    }

    catch( IOException e ) {
       System.err.println( "Problems opening file\n" + e.toString() );
       System.exit( 1 );
    }

    foo = new APMLexer( fileToScan );

  }

  public static Symbol next_token() throws java.io.IOException
  {
    return foo.yylex();
  }


%}


%class APMLexer
%type Symbol
%eofval{
  return ( new Symbol( APMParserSym.EOF ) );
%eofval}

CHAR = [a-zA-Z_]
QUOTED_TEXT = \"[^\"\r\n]*[\"\n]
WHITE_SPACES = [\ \t\b]+
WHITE_SPACE = [\ \t\b]
COMMENT = (\(\*)(([^(\*\))]*[\r\n]*)*)(\*\))
NEWLINE = [\r\n]+
ID = ([a-zA-Z][a-zA-Z0-9_]*\.)*[a-zA-Z][a-zA-Z0-9_]*
LOGICAL_OPERATOR = (\=\=)
COLON = :
SEMI_COLON = ;
LEFT_BRACKET = \[
RIGHT_BRACKET = \]
COMMA = \,
BOUND = [0-9\?]+
%%


"DOMAIN" { return new Symbol( APMParserSym.DOMAIN ); }
"SUBTYPE_OF" { return new Symbol( APMParserSym.SUBTYPE_OF ); }
"END_DOMAIN" { return new Symbol( APMParserSym.END_DOMAIN ); }
"MULTI_LEVEL_DOMAIN" { return new Symbol( APMParserSym.MULTI_LEVEL_DOMAIN ); }
"END_MULTI_LEVEL_DOMAIN" { return new Symbol( APMParserSym.END_MULTI_LEVEL_DOMAIN ); }
```

```
"LIST" { return new Symbol( APMParserSym.LIST );  }
"OF" { return new Symbol( APMParserSym.OF );  }
"MULTI_LEVEL" { return new Symbol( APMParserSym.MULTI_LEVEL );  }
"IDEALIZED" { return new Symbol( APMParserSym.IDEALIZED );  }
"ESSENTIAL" { return new Symbol( APMParserSym.ESSENTIAL );  }
"SOURCE_SET" { return new Symbol( APMParserSym.SOURCE_SET ); }
"ROOT_DOMAIN" { return new Symbol( APMParserSym.ROOT_DOMAIN ); }
"END_SOURCE_SET" { return new Symbol( APMParserSym.END_SOURCE_SET ); }
"PRODUCT_RELATIONS" { return new Symbol( APMParserSym.PRODUCT_RELATIONS ); }
"PRODUCT_IDEALIZATION_RELATIONS" { return new Symbol(
APMParserSym.PRODUCT_IDEALIZATION_RELATIONS ); }
"LINK_DEFINITIONS" { return new Symbol( APMParserSym.LINK_DEFINITIONS ); }
"END_LINK_DEFINITIONS" { return new Symbol( APMParserSym.END_LINK_DEFINITIONS ); }
{LOGICAL_OPERATOR} { return new Symbol( APMParserSym.LOGICAL_OPERATOR , yytext() ); }
"APM" { return new Symbol( APMParserSym.APM ); }
"END_APM" { return new Symbol( APMParserSym.END_APM ); }
"REAL" { return new Symbol( APMParserSym.REAL ); }
"STRING" { return new Symbol( APMParserSym.STRING ); }
{QUOTED_TEXT} { return new Symbol( APMParserSym.QUOTED_TEXT , yytext().substring( 1 , yytext().length() -1 ) ); }
{ID} { return new Symbol( APMParserSym.ID, yytext() ); }
{WHITE_SPACES} { }
{WHITE_SPACE} { }
{NEWLINE} {  }
{COMMENT} { }
{COLON} { return new Symbol( APMParserSym.COLON ); }
{SEMI_COLON} { return new Symbol( APMParserSym.SEMI_COLON ); }
{LEFT_BRACKET} { return new Symbol( APMParserSym.LEFT_BRACKET ); }
{RIGHT_BRACKET} { return new Symbol( APMParserSym.RIGHT_BRACKET ); }
{COMMA} { return new Symbol( APMParserSym.COMMA ); }
{BOUND} { return new Symbol( APMParserSym.BOUND , yytext() ); }
. { System.out.println( "Illegal character: <"  + yytext() + ">" ); }
```

Java-CUP APM Structure Definition Language Grammar Specification

```
import java_cup.runtime.Symbol;
import apm.*;
import java.util.*;



action code
{:

  /* Diego's additions under "action code" */

  /* Routines and variables for use by the code embedded in the grammar will normally
     be placed in this section (a typical example might be symbol table manipulation
     routines) */

  /* This code gets copied verbatim inside CUP$APMParser$actions class */


  /* The parser creates an instance of CUP$APMParser$actions and calls it action_obj */

  /* Variables used by the methods created in this section and the grammar actions */
  APMPrimitiveDomain realDomain , stringDomain;
  APMPrimitiveAggregateDomain listOfReals , listOfStrings;
  ListOfAPMAttributes tempListOfAPMAttributes;
  ListOfAPMDomains tempListOfAPMDomains;
  ListOfAPMSourceSets tempListOfAPMSourceSets;
  ListOfAPMRelations tempListOfAPMRelations;
  Hashtable tableOfDefinedAPMDomains;
  ListOfAPMSourceSetLinks tempListOfAPMSourceSetLinks;
  String domain1 , domain2;
  String set1 , set2;

  void initTables()
  {
    tempListOfAPMAttributes = new ListOfAPMAttributes();
    tempListOfAPMDomains = new ListOfAPMDomains();
    tempListOfAPMSourceSets = new ListOfAPMSourceSets();
    tempListOfAPMRelations = new ListOfAPMRelations();
    tableOfDefinedAPMDomains = new Hashtable();
    tempListOfAPMSourceSetLinks = new ListOfAPMSourceSetLinks();

    /* Create primitive domains */
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    APMPrimitiveDomain realDomain;
    APMPrimitiveDomain stringDomain;
    APMPrimitiveAggregateDomain listOfReals;
    APMPrimitiveAggregateDomain listOfStrings;

  }


  void setContainerDomain( ListOfAPMAttributes atts , APMComplexDomain dom )
  {
    for( int i = 0 ; i < atts.size() ; i++ )
      atts.elementAt( i ).setContainerDomain( dom );
  }

  boolean checkDependencies( String setName )
  {

    APMAttribute tempAPMAttribute;
    APMComplexAttribute tempAPMComplexAttribute;
    APMAggregateAttribute tempAPMAggregateAttribute;
    ListOfAPMAttributes tempListOfAPMAttributes = new ListOfAPMAttributes();
    String domainName;
    Object tempTableElement;
```

```
boolean problems = false;
APMDomain tempAPMDomain;
APMComplexDomain tempAPMComplexDomain;
APMObjectDomain tempAPMObjectDomain;
APMMultiLevelDomain tempAPMMultiLevelDomain;
APMAggregateDomain tempAPMAggregateDomain;
APMDomain domainOfElements;
String supertypeDomainName;

System.out.println( );
System.out.println( "Checking dependencies in source set \"" + setName + "\"..." );

/*
  Checks the following:
    1. That the domains of the attributes in each complex domain are defined
       (i.e., they are either valid complex, primitive or aggregate domains)
    2. That the domains used in aggregate domains are defined

  Does not check:
    1. Primitive domains (they are always created for all source sets, so it is
       safe not to check them)
*/

// Check each domain in table tableOfDefinedAPMDomains (except APMPrimitiveDomains)
for( Enumeration enum = tableOfDefinedAPMDomains.elements() ; enum.hasMoreElements() ; )
{

  /* Get a domain from the table */
  tempTableElement = enum.nextElement();

  /* Cast it to APMDomain */
  tempAPMDomain = (APMDomain) tempTableElement;

  /* If the domain is an APMComplexDomain , check its attributes (or levels) */
  if( tempAPMDomain.isAnAPMComplexDomain() )
  {
    /* Cast it to APMComplexDomain */
    tempAPMComplexDomain = (APMComplexDomain) tempTableElement;

    /* Get the attributes (if APMObjectDomain) or the levels (if MultiLevelDomain) */
    if( tempAPMComplexDomain.isAnAPMObjectDomain() )
    {
      /* Cast it to APMObjectDomain */
      tempAPMObjectDomain = (APMObjectDomain) tempAPMComplexDomain;

      /* If this APMObjectDomain is a subtype of another APMObjectDomain, check
         that the second has been indeed defined */
      if( tempAPMObjectDomain.hasSupertype() )
      {
        supertypeDomainName = tempAPMObjectDomain.getSupertypeDomain().getDomainName();
        if( ! tableOfDefinedAPMDomains.containsKey(  supertypeDomainName ) )
        {
          /* The domain has not been defined */
          System.err.println( "Supertype domain \"" + supertypeDomainName + "\" used in domain \"" +
            tempAPMObjectDomain.getDomainName() + "\" has not been defined" );
          problems = true;
        }
        else
        {
          /* The domain has been defined, redirect the dummy supertype attribute to the domain in the table */
          tempAPMObjectDomain.setSupertypeDomain( (APMObjectDomain) tableOfDefinedAPMDomains.get(
            supertypeDomainName ) );
        }

      }
```

```
  /* Next, get the list of attributes of this APMDomain to be checked later */
  tempListOfAPMAttributes = tempAPMObjectDomain.getLocalAttributes();

}

/* The complex domain is a multi-level domain */
else if( tempAPMComplexDomain.isAnAPMMultiLevelDomain() )
{
  /* Cast it to APMMultiLevelDomain */
  tempAPMMultiLevelDomain = (APMMultiLevelDomain) tempAPMComplexDomain;
  tempListOfAPMAttributes = tempAPMMultiLevelDomain.getLevels();
}

/* Now, check each attribute of this complex domain */
for( int i = 0 ; i < tempListOfAPMAttributes.size() ; i++ )
{
  // Get an attribute from the list of attributes
  tempAPMAttribute = tempListOfAPMAttributes.elementAt( i );

  // Get the name of its domain
  domainName = tempAPMAttribute.getDomain().getDomainName();

  // First, check that the domain of tempAPMAttribute has been defined
  if( tableOfDefinedAPMDomains.containsKey( domainName ) )
  {
    // The domain of tempAPMAttribute is in the table of defined domains.

    // At this point, the domain of tempAPMAttribute is a dummy domain.
    // We have to redirect it to the domain stored in the table

    if( tempAPMAttribute.isAnAPMComplexAttribute() )
    {
      /* The attribute is an APMComplexAttribute, cast it */
      tempAPMComplexAttribute = (APMComplexAttribute) tempListOfAPMAttributes.elementAt( i );

      if( tempAPMComplexAttribute.isAnAPMObjectAttribute() )
      {
        /*
        Check that we haven't decared this attribute as an APMObjectAttribute
        when its domain is an APMMultiLevelDomain
        */
        if( ( (APMComplexDomain) tableOfDefinedAPMDomains.get( domainName )
            ).isAnAPMMultiLevelDomain() )
        {
          System.err.println( "ERROR: Domain \"" + domainName + "\" is a multi-level domain." );
          System.err.println( "Attribute \"" + tempAPMComplexAttribute.getAttributeName() + "\" must be defined
            as a multi_level_attribute" );
          problems = true;
        }

        // Everything is OK, redirect the reference to the complex domain in the table
        tempAPMComplexAttribute.setDomain( (APMObjectDomain) tableOfDefinedAPMDomains.get(
          domainName ) );
      }

      else if( tempAPMComplexAttribute.isAnAPMMultiLevelAttribute() )
        tempAPMComplexAttribute.setDomain( (APMMultiLevelDomain) tableOfDefinedAPMDomains.get(
          domainName ) );

    }

    else if( tempListOfAPMAttributes.elementAt( i ).isAnAPMAggregateAttribute() )
    {

      // The domain is an aggregate domain. Need to check its domain
```

```
            /* The attribute is an APMAggregateAttribute, cast it */
            tempAPMAggregateAttribute = (APMAggregateAttribute) tempListOfAPMAttributes.elementAt( i );

            // Redirect the reference to the complex domain in the table
            tempAPMAggregateAttribute.setDomain( (APMAggregateDomain) tableOfDefinedAPMDomains.get(
              domainName ) );
          }
          else if( tempListOfAPMAttributes.elementAt( i ).isAnAPMPrimitiveAttribute() )
          {
            // The domain of the attribute is primitive. No need to check it.
          }


        } // End of if domainName is in the table

        // The domain is not in the table
        else if( ! tableOfDefinedAPMDomains.containsKey( domainName ) )
        {
          // The domain is not in the table (undefined domain)
          System.out.println( "ERROR: Domain \"" + domainName + "\" has not been defined in source set \"" +
            setName + "\"" );
          problems = true;
        }

      } // End loop for each attribute in tempListOfAttributes

    } // End if tempAPMDomain.isAnAPMComplexDomain


    else if( tempAPMDomain.isAnAPMAggregateDomain() )
    {
      /* The domain is an aggregate domain */
      /* Must check that the domain of the elements is OK */

      /* Cast it to APMAggregateDomain */
      tempAPMAggregateDomain = (APMAggregateDomain) tempAPMDomain;
      domainOfElements = tempAPMAggregateDomain.getDomainOfElements();

      /* Check if domainOfElements is OK */
      domainName = domainOfElements.getDomainName();

      /* The following line is to get the actual domain of the elements.
        (before, we had a dummy APMObjectDomain, when the actual domain
        of the elements can also be an APMPrimitiveDomain, APMMultiLevelDomain,
        or even another APMAggregateDomain)  */
      domainOfElements = (APMDomain) tableOfDefinedAPMDomains.get( domainName );

      if( tableOfDefinedAPMDomains.containsKey( domainName ) )
      {
        /* The domain of the elements is OK, point to it */

        if( domainOfElements.isAnAPMPrimitiveDomain() )
          ( (APMPrimitiveAggregateDomain) tempAPMAggregateDomain ).setDomainOfElements(
            (APMPrimitiveDomain) tableOfDefinedAPMDomains.get( domainName ) );
        else if( domainOfElements.isAnAPMObjectDomain() )
          ( (APMComplexAggregateDomain) tempAPMAggregateDomain ).setDomainOfElements( (APMObjectDomain)
            tableOfDefinedAPMDomains.get( domainName ) );
        else if( domainOfElements.isAnAPMMultiLevelDomain() )
          ( (APMComplexAggregateDomain) tempAPMAggregateDomain ).setDomainOfElements(
            (APMMultiLevelDomain) tableOfDefinedAPMDomains.get( domainName ) );
      }

      else
      {
        /* The domain is not OK */
        System.out.println( "ERROR: Domain \"" + domainName + "\" has not been defined" );
```

```
            problems = true;
          }
        }

    } // End loop for each element in tableOfDefinedAPMDomains

    if( !problems )
    {
      System.out.println( "Domain dependencies in source set \"" + setName + "\" OK." );
      return true;
    }
    else
    {
      System.out.println( "ERROR: Incorrect domain reference in source set \"" + setName + "\"." );
      return false;
    }

  }


  void setSourceSet( APMSourceSet set , ListOfAPMDomains domains )
  {
    for( int i = 0 ; i < domains.size() ; i++ )
      domains.elementAt( i ).setSourceSet( set );
  }

  ListOfStrings stripFullAttributeName( String fullAttributeName )
  {


    /* Takes a chain of dot-separated strings and creates a list of
       strings with each individual piece of text */

    /* Example: Takes "a.b.c" and returns the list: { "a" , "b" , "c" } */

    StringTokenizer tokens = new StringTokenizer( fullAttributeName , "." );
    ListOfStrings returnList = new ListOfStrings() ;

    while( tokens.hasMoreTokens() )
      returnList.addElement( tokens.nextToken() );

    return returnList;
  }


  APMDomain getDomainFromList( String domainName , ListOfAPMDomains listOfAPMDomains )
  {
    APMDomain tempAPMDomain;

    for( int i = 0 ; i < listOfAPMDomains.size() ; i++ )
    {
      tempAPMDomain = listOfAPMDomains.elementAt( i );
      if( tempAPMDomain.getDomainName().equalsIgnoreCase( domainName ) )
        return tempAPMDomain;
    }

    return null;
  }



:} ;

parser code
{:
  /* Diego's additions under "parser code" */
```

```
  /* This code gets copied verbatim inside APMParser class */

  /* The code in this section is placed directly within the generated parser class.
     Although this is less common, it can be helpful when customizinf the parser. It is
     possible, for example, to inlude scanning methods inside the parsser and or override
     the default error reporting routines */


  /* New member variable */
  String fileToParse = "input.txt"; // Default

  /* Overload the default constructor so that we can specify
     the name of the file to parse */
  public APMParser( String fileName )
  {
    super();
    fileToParse = fileName;
  }

:};

init with
{:
  /* Diego's additions under "init with" */
  /* This code gets copied verbatim inside APMParser.user_init() method */

  /* The code in this section will be executed by the parser before it
     asks for the first token. Typically, this is used to initialize the
     scanner as well as various tables and other data structures that
     might be needed by the semantic actions. */

  /*
  NOTE: If need to call methods belonging to class CUP$APMParser$actions (like
  the methods defined in the "action code" section above) use the action_obj
  (which is an instance of CUP$APMParser$actions created by the APMParser)
  */

  action_obj.initTables();
  APMLexer.init( fileToParse );
:} ;

scan with
{:

  /* Indicates how th parser should ask for the next token from the scanner */
  return APMLexer.next_token();

:} ;



terminal APM ,
        END_APM,
        SOURCE_SET,
        ROOT_DOMAIN ,
        END_SOURCE_SET,
        DOMAIN ,
        END_DOMAIN ,
        SUBTYPE_OF ,
        MULTI_LEVEL_DOMAIN ,
        END_MULTI_LEVEL_DOMAIN ,
        LEFT_BRACKET ,
        RIGHT_BRACKET ,
        COMMA ,
        LIST ,
        OF ,
```

```
        MULTI_LEVEL ,
        IDEALIZED ,
        ESSENTIAL ,
        PRODUCT_RELATIONS ,
        PRODUCT_IDEALIZATION_RELATIONS ,
        REAL ,
        STRING ,
        COLON ,
        SEMI_COLON,
        LINK_DEFINITIONS ,
        END_LINK_DEFINITIONS;

terminal String ID;
terminal String QUOTED_TEXT;
terminal String LOGICAL_OPERATOR;
terminal String BOUND;

non terminal APMAttribute attribute;
non terminal APM apm_definition;
non terminal source_set_definitions;
non terminal APMSourceSet source_set_definition;


non terminal domains;

non terminal APMDomain domain;

non terminal attributes;
non terminal relations;
non terminal listOfProductIdealizationRelations;
non terminal listOfProductRelations;
non terminal productRelations;
non terminal productIdealizationRelations;
non terminal APMProductRelation productRelation;
non terminal APMProductIdealizationRelation productIdealizationRelation;

non terminal link_definitions;
non terminal link_defs;
non terminal APMSourceSetLink link_def;




apm_definition ::=
  APM ID:t1 SEMI_COLON source_set_definitions link_definitions END_APM SEMI_COLON
  {:
    RESULT = new APM( t1 , tempListOfAPMSourceSets , tempListOfAPMSourceSetLinks );
    System.out.println( "Finished parsing APM Definition \"" + t1 + "\"" );
  :}
  |
  APM ID:t1 SEMI_COLON source_set_definitions END_APM SEMI_COLON
  {:
    RESULT = new APM( t1 , tempListOfAPMSourceSets );
    System.out.println( "Finished parsing APM Definition \"" + t1 + "\"" );
  :} ;

source_set_definitions ::= source_set_definition | source_set_definitions source_set_definition;


source_set_definition ::=
  SOURCE_SET ID:t1 ROOT_DOMAIN ID:t2 SEMI_COLON domains END_SOURCE_SET SEMI_COLON
  {:

    // Add the primitive domains to the list of domains of this source set
```

```
    // Also add them to the table of defined domains
    if( realDomain != null )
    {
      tempListOfAPMDomains.addElement( realDomain );
      tableOfDefinedAPMDomains.put( "REAL" , realDomain );
    }
    if( stringDomain != null )
    {
      tempListOfAPMDomains.addElement( stringDomain );
      tableOfDefinedAPMDomains.put( "STRING" , stringDomain );
    }
    if( listOfReals != null )
    {
      tempListOfAPMDomains.addElement( listOfReals );
      tableOfDefinedAPMDomains.put( "ListOfReals" , listOfReals );
    }
    if( listOfStrings != null )
    {
      tempListOfAPMDomains.addElement( listOfStrings );
      tableOfDefinedAPMDomains.put( "ListOfStrings" , listOfStrings );
    }
    // Clear the domains for the next source set
    realDomain = null;
    stringDomain = null;
    listOfReals = null;
    listOfStrings = null;



    // Check the OID references made within the complex domains in this set
    if( ! checkDependencies( t1 ) )
      return null;

    else  // Dependencies OK
    {
      // Get a reference to the root domain
      APMComplexDomain rootDomain = (APMComplexDomain) getDomainFromList( t2 , tempListOfAPMDomains );

      // Create the source set
      RESULT = new APMSourceSet( t1 , tempListOfAPMDomains , rootDomain );
      System.out.println( "Finished parsing definitions in source set \"" + t1 + "\"" );
      System.out.println( );
      tempListOfAPMSourceSets.addElement( RESULT );

      // Set the sourceSet attribute of the domains contained in this source set
      setSourceSet( RESULT , tempListOfAPMDomains );

      // Clear lists and tables
      tempListOfAPMDomains.empty();
      tableOfDefinedAPMDomains.clear( );
    }

    System.out.println( "Parsed source set \"" + t1 + "\"" );
  :};

domains ::= domain | domains domain;


domain ::=
  DOMAIN ID:t1 SEMI_COLON attributes END_DOMAIN SEMI_COLON
  {:
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    RESULT = new APMObjectDomain( t1 , tempListOfAPMAttributes , dummySourceSet );
    setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
    tableOfDefinedAPMDomains.put( t1 , RESULT );
```

```
  tempListOfAPMDomains.addElement( RESULT );
  tempListOfAPMAttributes.empty();
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SUBTYPE_OF ID:t2 SEMI_COLON attributes END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummySupertypeDomain = new APMObjectDomain( t2 , dummySourceSet );
  RESULT = new APMObjectDomain( t1 , dummySupertypeDomain , tempListOfAPMAttributes , dummySourceSet );
  setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
  tableOfDefinedAPMDomains.put( t1 , RESULT );
  tempListOfAPMDomains.addElement( RESULT );
  tempListOfAPMAttributes.empty();
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SEMI_COLON  attributes relations END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  RESULT = new APMObjectDomain( t1 , tempListOfAPMAttributes , tempListOfAPMRelations , dummySourceSet );
  setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
  tableOfDefinedAPMDomains.put( t1 , RESULT );
  tempListOfAPMDomains.addElement( RESULT );
  tempListOfAPMAttributes.empty();
  tempListOfAPMRelations.empty();
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SUBTYPE_OF ID:t2 SEMI_COLON attributes relations END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummySupertypeDomain = new APMObjectDomain( t2 , dummySourceSet );
  RESULT = new APMObjectDomain( t1 , dummySupertypeDomain , tempListOfAPMAttributes ,
    tempListOfAPMRelations , dummySourceSet );
  setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
  tableOfDefinedAPMDomains.put( t1 , RESULT );
  tempListOfAPMDomains.addElement( RESULT );
  tempListOfAPMAttributes.empty();
  tempListOfAPMRelations.empty();
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SUBTYPE_OF ID:t2 SEMI_COLON relations END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummySupertypeDomain = new APMObjectDomain( t2 , dummySourceSet );
  RESULT = new APMObjectDomain( t1 , dummySupertypeDomain , tempListOfAPMRelations , dummySourceSet );
  tableOfDefinedAPMDomains.put( t1 , RESULT );
  tempListOfAPMDomains.addElement( RESULT );
  tempListOfAPMRelations.empty();
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SEMI_COLON END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  RESULT = new APMObjectDomain( t1 , dummySourceSet );
  tableOfDefinedAPMDomains.put( t1 , RESULT );
  tempListOfAPMDomains.addElement( RESULT );
  System.out.println( "Parsed domain \"" + t1 + "\"" );
:}
|
DOMAIN ID:t1 SUBTYPE_OF ID:t2 SEMI_COLON END_DOMAIN SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
```

```
    APMObjectDomain dummySupertypeDomain = new APMObjectDomain( t2 , dummySourceSet );
    RESULT = new APMObjectDomain( t1 , dummySupertypeDomain , dummySourceSet );
    tableOfDefinedAPMDomains.put( t1 , RESULT );
    tempListOfAPMDomains.addElement( RESULT );
    System.out.println( "Parsed domain \"" + t1 + "\"" );
  :}
  |
  MULTI_LEVEL_DOMAIN ID:t1 SEMI_COLON attributes END_MULTI_LEVEL_DOMAIN SEMI_COLON
  {:
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    RESULT = new APMMultiLevelDomain( t1 , tempListOfAPMAttributes , dummySourceSet );
    setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
    tempListOfAPMDomains.addElement( RESULT );
    tableOfDefinedAPMDomains.put( t1 , RESULT );
    tempListOfAPMAttributes.empty();
    System.out.println( "Parsed domain \"" + t1 + "\"" );
  :}
  |
  MULTI_LEVEL_DOMAIN ID:t1 SEMI_COLON attributes relations END_MULTI_LEVEL_DOMAIN SEMI_COLON
  {:
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    RESULT = new APMMultiLevelDomain( t1 , tempListOfAPMAttributes , tempListOfAPMRelations ,
      dummySourceSet );
    setContainerDomain( tempListOfAPMAttributes , (APMComplexDomain) RESULT );
    tempListOfAPMDomains.addElement( RESULT );
    tableOfDefinedAPMDomains.put( t1 , RESULT );
    tempListOfAPMAttributes.empty();
    tempListOfAPMRelations.empty();
    System.out.println( "Parsed domain \"" + t1 + "\"" );
  :};



attributes ::= attribute | attributes attribute ;

attribute ::=
  ID:t1 COLON REAL SEMI_COLON
  {:
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

    // Create the real domain (if it hasn't been created)
      if( realDomain == null )
        realDomain = new APMPrimitiveDomain( "REAL" , dummySourceSet );

    RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.PRODUCT ,
      realDomain );
    tempListOfAPMAttributes.addElement( RESULT );
  :}
  |
  ID:t1 COLON STRING SEMI_COLON
  {:
    APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
    APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

    // Create the string domain (if it hasn't been created)
    if( stringDomain == null )
      stringDomain = new APMPrimitiveDomain( "STRING" , dummySourceSet );

    RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.PRODUCT ,
      stringDomain );
    tempListOfAPMAttributes.addElement( RESULT );
  :}
  |
  ESSENTIAL ID:t1 COLON REAL SEMI_COLON
  {:
```

```
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the real domain (if it hasn't been created)
  if( realDomain == null )
    realDomain = new APMPrimitiveDomain( "REAL" , dummySourceSet );

  RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.ESSENTIAL ,
    realDomain );

  tempListOfAPMAttributes.addElement( RESULT );

:}
|
ESSENTIAL ID:t1 COLON STRING SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the string domain (if it hasn't been created)
  if( stringDomain == null )
    stringDomain = new APMPrimitiveDomain( "STRING" , dummySourceSet );

  RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.ESSENTIAL ,
    stringDomain );

  tempListOfAPMAttributes.addElement( RESULT );

:}
|
IDEALIZED ID:t1 COLON REAL SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the real domain (if it hasn't been created)
  if( realDomain == null )
    realDomain = new APMPrimitiveDomain( "REAL" , dummySourceSet );

  RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.IDEALIZED ,
    realDomain );

  tempListOfAPMAttributes.addElement( RESULT );

:}
|
IDEALIZED ID:t1 COLON STRING SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the string domain (if it hasn't been created)
  if( stringDomain == null )
    stringDomain = new APMPrimitiveDomain( "STRING" , dummySourceSet );

  RESULT =  new APMPrimitiveAttribute( t1 , dummyContainerDomain , APMPrimitiveAttribute.IDEALIZED ,
    stringDomain );

  tempListOfAPMAttributes.addElement( RESULT );

:}
|
ID:t1 COLON ID:t2 SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyDomain = new APMObjectDomain( t2 , dummySourceSet );
```

```
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );
  RESULT =  new APMObjectAttribute( t1 , dummyContainerDomain , dummyDomain );
  tempListOfAPMAttributes.addElement( RESULT );
:}
|
ID:t1 COLON MULTI_LEVEL ID:t2 SEMI_COLON
{:

  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMMultiLevelDomain dummyDomain = new APMMultiLevelDomain( t2 , dummySourceSet );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );
  RESULT =  new APMMultiLevelAttribute( t1 , dummyContainerDomain , dummyDomain );
  tempListOfAPMAttributes.addElement( RESULT );

:}
|
ID:t1 COLON LIST LEFT_BRACKET BOUND:b1 COMMA BOUND:b2 RIGHT_BRACKET OF ID:t2 SEMI_COLON
{:
  // Implicitly create the domain (i.e., user does not have to define it)
  // Using a domain name consisting of t2 prefixed with "ListOf" and sufixed with "s"
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyDomainOfElements = new APMObjectDomain( t2 , dummySourceSet );
  APMComplexAggregateDomain newAggregateDomain = new APMComplexAggregateDomain( "ListOf" + t2 + "s" ,
    dummySourceSet , dummyDomainOfElements );
  tempListOfAPMDomains.addElement( newAggregateDomain );
  tableOfDefinedAPMDomains.put( "ListOf" + t2 + "s" , newAggregateDomain );

  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );
  RESULT = new APMComplexAggregateAttribute( t1 , dummyContainerDomain , newAggregateDomain , b1 , b2 );
  tempListOfAPMAttributes.addElement( RESULT );
:}
|
ID:t1 COLON LIST LEFT_BRACKET BOUND:b1 COMMA BOUND:b2 RIGHT_BRACKET OF REAL SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the listOfReals domain (if it hasn't been created)
  if( realDomain == null )
    realDomain = new APMPrimitiveDomain( "REAL" , dummySourceSet );

  if( listOfReals == null )
    listOfReals = new APMPrimitiveAggregateDomain( "ListOfReals" , dummySourceSet , realDomain );

  RESULT = new APMPrimitiveAggregateAttribute( t1 , dummyContainerDomain , listOfReals , b1 , b2 );

  tempListOfAPMAttributes.addElement( RESULT );
:}
|
ID:t1 COLON LIST LEFT_BRACKET BOUND:b1 COMMA BOUND:b2 RIGHT_BRACKET OF STRING SEMI_COLON
{:
  APMSourceSet dummySourceSet = new APMSourceSet( "dummySourceSet" );
  APMObjectDomain dummyContainerDomain = new APMObjectDomain( "Don't know yet" , dummySourceSet );

  // Create the listOfStrings domain (if it hasn't been created)
  if( stringDomain == null )
    stringDomain = new APMPrimitiveDomain( "STRING" , dummySourceSet );

  if( listOfStrings == null )
    listOfStrings = new APMPrimitiveAggregateDomain( "ListOfStrings" , dummySourceSet , stringDomain );

  RESULT = new APMPrimitiveAggregateAttribute( t1 , dummyContainerDomain , listOfStrings , b1 , b2 );

  tempListOfAPMAttributes.addElement( RESULT );
:} ;
```

```
relations ::= productIdealizationRelations productRelations |
            productRelations productIdealizationRelations |
            productIdealizationRelations |
            productRelations
            ;

productIdealizationRelations ::= PRODUCT_IDEALIZATION_RELATIONS listOfProductIdealizationRelations;

listOfProductIdealizationRelations ::= productIdealizationRelation |
                                listOfProductIdealizationRelations productIdealizationRelation;

productIdealizationRelation ::= ID:t1 COLON QUOTED_TEXT:t2 SEMI_COLON
  {:
    RESULT = new APMProductIdealizationRelation( t1 , t2 );
    tempListOfAPMRelations.addElement( RESULT );
  :};


productRelations ::= PRODUCT_RELATIONS listOfProductRelations;

listOfProductRelations ::= productRelation |
                    listOfProductRelations productRelation;

productRelation ::= ID:t1 COLON QUOTED_TEXT:t2 SEMI_COLON
  {:
    RESULT = new APMProductRelation( t1 , t2 );
    tempListOfAPMRelations.addElement( RESULT );
  :} ;



link_definitions ::= LINK_DEFINITIONS link_defs END_LINK_DEFINITIONS SEMI_COLON
  {:
  :};


link_defs ::= link_def | link_defs link_def;


link_def ::= ID:t1 LOGICAL_OPERATOR:t2 ID:t3 SEMI_COLON
  {:

    ListOfStrings fullAttributeName1 = stripFullAttributeName( t1 );
    ListOfStrings fullAttributeName2 = stripFullAttributeName( t3 );

    APMSourceSetLinkAttribute key1 = new APMSourceSetLinkAttribute( fullAttributeName1 );
    APMSourceSetLinkAttribute key2 = new APMSourceSetLinkAttribute( fullAttributeName2 );

    RESULT = new APMSourceSetLink( key1 , key2 , t2 );

    tempListOfAPMSourceSetLinks.addElement( RESULT );

  :};
```

F.1 APM Instance Definition Language

Jlex APM Instance Definition Language Lexer Specification

```
package apm.wrapper;
import java.io.*;
import java_cup.runtime.Symbol;
import apm.wrapper.APMInstanceParserSym;

%%

%{
  static APMInstanceLexer foo;
  static BufferedReader fileToScan;

  public static void init( String fileName )
  {


    try {
       fileToScan = new BufferedReader( new FileReader( fileName ) );
    }

    catch( IOException e ) {
       System.err.println( "Problems opening file\n" + e.toString() );
       System.exit( 1 );
    }

    foo = new APMInstanceLexer( fileToScan );

  }

  public static Symbol next_token() throws java.io.IOException
  {
    return foo.yylex();
  }


%}


%class APMInstanceLexer
%type Symbol
%eofval{
  return ( new Symbol( APMInstanceParserSym.EOF ) );
%eofval}

REAL = -?(([0-9]+)|([0-9]*\.[0-9]+)([eE][\-+]?[0-9]+)?)
QUOTED_TEXT = \"[^\"\r\n]*[\"\n]
WHITE_SPACES = [\ \t\b]+
WHITE_SPACE = [\ \t\b]
NEWLINE = [\r\n]+
ID = ([a-zA-Z][a-zA-Z0-9_\[\]]*\.)*[a-zA-Z\[\]][a-zA-Z0-9_\[\]]*
COLON = :
SEMICOLON = ;
QUESTION_MARK = \?
%%


"DATA" { return new Symbol( APMInstanceParserSym.DATA ); }
"END_DATA" { return new Symbol( APMInstanceParserSym.END_DATA ); }
"INSTANCE_OF" { return new Symbol( APMInstanceParserSym.INSTANCE_OF ); }
"END_INSTANCE" { return new Symbol( APMInstanceParserSym.END_INSTANCE ); }
{COLON} { return new Symbol( APMInstanceParserSym.COLON ); }
{SEMICOLON} { return new Symbol( APMInstanceParserSym.SEMICOLON ); }
{QUESTION_MARK} { return new Symbol( APMInstanceParserSym.QUESTION_MARK ); }
{QUOTED_TEXT} { return new Symbol( APMInstanceParserSym.QUOTED_TEXT , yytext().substring( 1 , yytext().length()
-1 ) ); }
{ID} { return new Symbol( APMInstanceParserSym.ID , yytext() ); }
{REAL} { return new Symbol( APMInstanceParserSym.REAL , yytext() ); }
```

```
{WHITE_SPACES} { }
{WHITE_SPACE} { }
{NEWLINE} {  }

. { System.out.println( "Illegal character: <"  + yytext() + ">" ); }
```

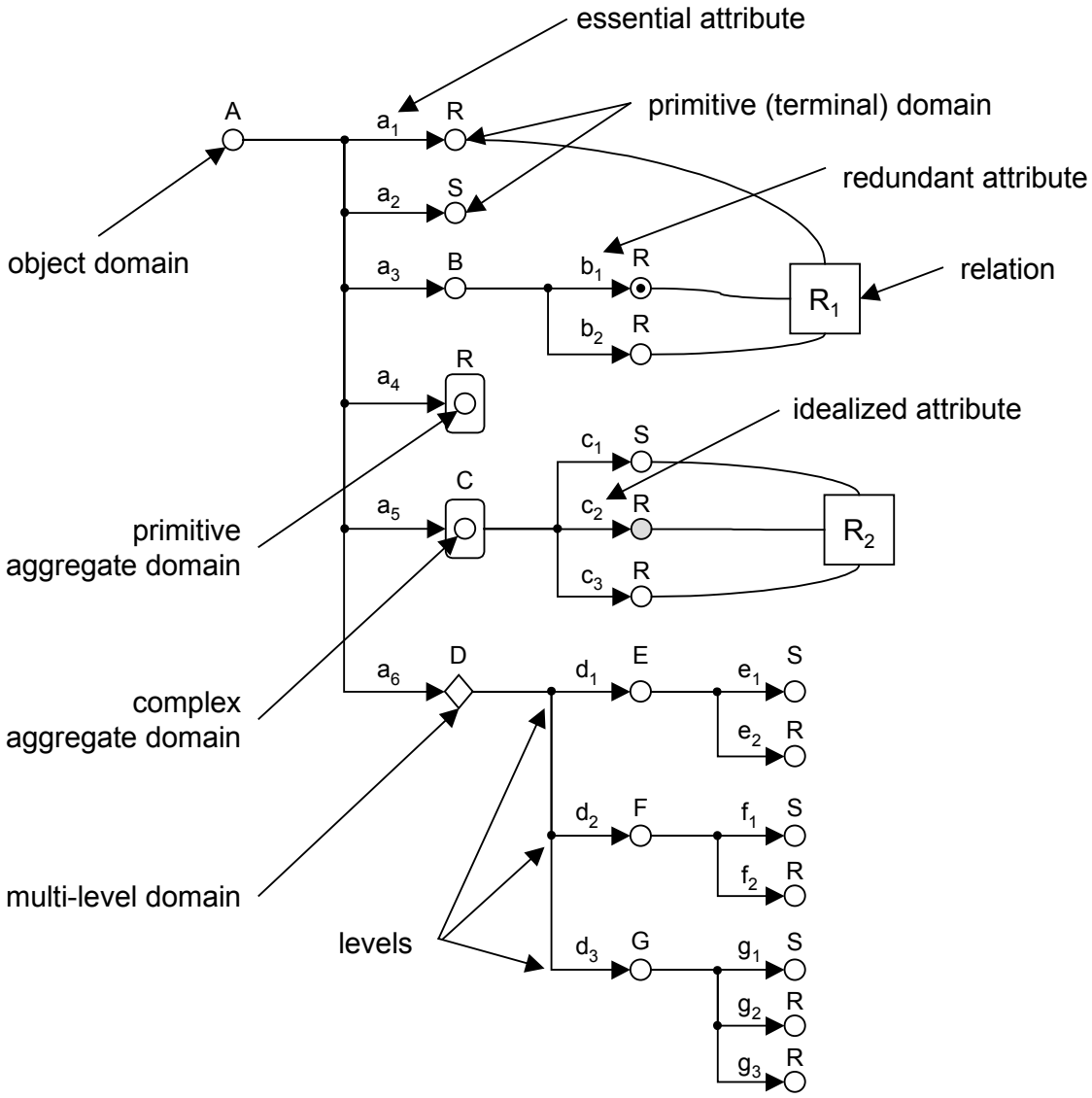
Java-CUP APM Instance Definition Language Grammar Specification

```
import java_cup.runtime.Symbol;
import apm.wrapper.*;

action code
{:

  ListOfAPMSourceDataWrapperReturnedObjects listOfAPMSourceDataWrapperReturnedObjects;
  ListOfAPMSourceDataWrapperReturnedValues listOfAPMSourceDataWrapperReturnedValues;

  void initTables()
  {
    listOfAPMSourceDataWrapperReturnedObjects = new ListOfAPMSourceDataWrapperReturnedObjects();
    listOfAPMSourceDataWrapperReturnedValues = new ListOfAPMSourceDataWrapperReturnedValues();
  }

:} ;

parser code
{:
  String fileToParse;

  /* Overload the default constructor so that we can specify
    the name of the file to parse */
  public APMInstanceParser( String fileName )
  {
    super();
    fileToParse = fileName;
  }

:};

init with
{:
    action_obj.initTables(  );
    APMInstanceLexer.init( fileToParse );
:} ;


scan with
{:
  return APMInstanceLexer.next_token();

:} ;



terminal DATA ,
       END_DATA ,
       INSTANCE_OF,
       END_INSTANCE ,
       COLON ,
       SEMICOLON ,
       QUESTION_MARK;

terminal String ID;
terminal String REAL;
terminal String QUOTED_TEXT;

non terminal ListOfAPMSourceDataWrapperReturnedObjects object_list;
non terminal objects;
non terminal APMSourceDataWrapperReturnedObject object;
non terminal values;
non terminal APMSourceDataWrapperReturnedValue value;

object_list ::= DATA SEMICOLON objects END_DATA SEMICOLON
  {:
```

```
    RESULT = listOfAPMSourceDataWrapperReturnedObjects;
  :};


objects ::= object | objects object;

object ::= INSTANCE_OF ID:t1 SEMICOLON values END_INSTANCE SEMICOLON
  {:

    RESULT = new APMSourceDataWrapperReturnedObject( t1 , listOfAPMSourceDataWrapperReturnedValues );
    listOfAPMSourceDataWrapperReturnedObjects.addElement( RESULT );
    listOfAPMSourceDataWrapperReturnedValues = new ListOfAPMSourceDataWrapperReturnedValues();

  :} ;


values ::= value | values value;

value ::=
  ID:t1 COLON QUOTED_TEXT:t2 SEMICOLON
  {:
    RESULT = new APMSourceDataWrapperReturnedStringValue( t1 , t2 );
    listOfAPMSourceDataWrapperReturnedValues.addElement( RESULT );
  :}
  |
  ID:t1 COLON REAL:t2 SEMICOLON
  {:
    RESULT = new APMSourceDataWrapperReturnedRealValue( t1 , Double.valueOf( t2 ).doubleValue() );
    listOfAPMSourceDataWrapperReturnedValues.addElement( RESULT );
  :}
  |
  ID:t1 COLON QUESTION_MARK SEMICOLON
  {:
    RESULT = new APMSourceDataWrapperReturnedNullValue( t1  );
    listOfAPMSourceDataWrapperReturnedValues.addElement( RESULT );
  :};
```

APPENDIX D

BASIC CONSTRAINT SCHEMATICS DIAGRAMS NOTATION


essential attribute
primitive (terminal) domain
redundant attribute
relation
idealized attribute
object domain
primitive
aggregate domain
complex
aggregate domain
multi-level domain
levels
A
a1
R
a2
S
a3
B
b1
R
b2
R
R1
a4
R
C
a5
c1
S
c2
R
c3
R
R2
D
a6
d1
E
e1
S
e2
R
d2
F
f1
S
f2
R
d3
G
g1
S
g2
R
g3
R

APPENDIX E

EXPRESS APM INFORMATION MODEL

```
SCHEMA apm_schema;

(* APM Definitions *)

ENTITY apm;
   name : STRING;
   source_sets : LIST[0:?] OF apm_source_set;
   source_set_links : LIST[0:?] OF apm_source_set_link;
   linked_domains : LIST[0:?] OF apm_domain;
   linked_instances : LIST[0:?] OF apm_complex_domain_instance;
   constraint_network : constraint_network;
END_ENTITY;

ENTITY apm_interface;
   name : STRING;
   active_apm : apm;
   list_of_apms : LIST[0:?] OF apm;
END_ENTITY;

ENTITY apm_source_set;
   source_set_name : STRING;
   domains_in_set : LIST[0:?] OF apm_domain;
   set_instances : LIST[0:?] OF apm_complex_domain_instance;
   source_set_data_wrapper : OPTIONAL apm_source_data_wrapper_object;
   data_repository_name : OPTIONAL STRING;
   root_domain : OPTIONAL apm_complex_domain;
END_ENTITY;

ENTITY apm_source_set_link;
   key_attribute_1 : apm_source_set_link_attribute;
   key_attribute_2 : apm_source_set_link_attribute;
   logical_operator : STRING;
END_ENTITY;

ENTITY apm_source_set_link_attribute;
   full_attribute_name : LIST[1:?] OF STRING;
END_ENTITY;


(* APM Domain Definitions *)

ENTITY apm_domain
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_domain , apm_primitive_domain ,
                                 apm_aggregate_domain ) );
   domain_name : STRING;
   domain_description : OPTIONAL STRING;
   source_set : apm_source_set;
END_ENTITY;

ENTITY apm_complex_domain
   ABSTRACT SUPERTYPE OF (ONEOF( apm_object_domain , apm_multi_level_domain ) )
   SUBTYPE OF( apm_domain );
   local_relations : LIST[0:?] OF apm_relation;
END_ENTITY;
```

```
ENTITY apm_object_domain
   SUBTYPE OF( apm_complex_domain );
   supertype_domain : OPTIONAL apm_object_domain;
   local_attributes : LIST[0:?] OF apm_attribute;
END_ENTITY;

ENTITY apm_multi_level_domain
   SUBTYPE OF( apm_complex_domain );
   levels : LIST[0:?] OF apm_attribute;
END_ENTITY;

ENTITY apm_primitive_domain
   SUBTYPE OF( apm_domain );
   (* WHERE rule to make sure that domain_name is "real" , "string" , etc.  *)
END_ENTITY;

ENTITY apm_aggregate_domain
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_aggregate_domain ,
apm_primitive_aggregate_domain ) )
   SUBTYPE OF( apm_domain );
END_ENTITY;

ENTITY apm_complex_aggregate_domain
   SUBTYPE OF( apm_aggregate_domain );
   domain_of_elements : apm_complex_domain;
END_ENTITY;

ENTITY apm_primitive_aggregate_domain
   SUBTYPE OF( apm_aggregate_domain );
   domain_of_elements : apm_primitive_domain;
END_ENTITY;


(* APM Attribute Definitions *)

ENTITY apm_attribute
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_attribute , apm_primitive_attribute ,
                                 apm_aggregate_attribute ) );
   attribute_name : STRING;
   attribute_description : OPTIONAL STRING;
   container_domain : apm_complex_domain;
END_ENTITY;

ENTITY apm_aggregate_attribute
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_aggregate_attribute ,
apm_primitive_aggregate_attribute ) )
   SUBTYPE OF( apm_attribute );
   low_bound : STRING;
   high_bound : STRING;
END_ENTITY;

ENTITY apm_complex_aggregate_attribute
   SUBTYPE OF( apm_aggregate_attribute );
   domain : apm_complex_aggregate_domain;
END_ENTITY;
```

```
ENTITY apm_primitive_aggregate_attribute
   SUBTYPE OF( apm_aggregate_attribute );
   domain : apm_primitive_aggregate_domain;
END_ENTITY;

ENTITY apm_complex_attribute
   ABSTRACT SUPERTYPE OF (ONEOF( apm_object_attribute , apm_multi_level_attribute ) )
   SUBTYPE OF( apm_attribute );
END_ENTITY;

ENTITY apm_object_attribute
   SUBTYPE OF( apm_complex_attribute );
   domain : apm_object_domain;
END_ENTITY;

ENTITY apm_multi_level_attribute
   SUBTYPE OF( apm_complex_attribute );
   domain : apm_multi_level_domain;
END_ENTITY;

ENTITY apm_primitive_attribute
   SUBTYPE OF( apm_attribute );
   category : INTEGER;
   domain : apm_primitive_domain;
END_ENTITY;


(* APM Instance Definitions *)

ENTITY apm_domain_instance
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_domain_instance ,
                                 apm_primitive_domain_instance ,
                                 apm_aggregate_domain_instance ) );
   attribute_name : STRING;
   contained_in : OPTIONAL apm_complex_domain_instance;
   element_of : OPTIONAL apm_aggregate_domain_instance;
END_ENTITY;

ENTITY apm_aggregate_domain_instance
   ABSTRACT SUPERTYPE OF (ONEOF( apm_complex_aggregate_domain_instance ,
apm_primitive_aggregate_domain_instance ) )
   SUBTYPE OF ( apm_domain_instance );
END_ENTITY;

ENTITY apm_complex_aggregate_domain_instance
   SUBTYPE OF ( apm_aggregate_domain_instance );
   elements : LIST[0:?] OF apm_complex_domain_instance;
   domain : apm_complex_aggregate_domain;
END_ENTITY;

ENTITY apm_primitive_aggregate_domain_instance
   SUBTYPE OF ( apm_aggregate_domain_instance );
   elements : LIST[0:?] OF apm_primitive_domain_instance;
   domain : apm_primitive_aggregate_domain;
END_ENTITY;
```

```
ENTITY apm_complex_domain_instance
   ABSTRACT SUPERTYPE OF (ONEOF( apm_object_domain_instance ,
                                 apm_multi_level_domain_instance ) )
   SUBTYPE OF ( apm_domain_instance );
   values : LIST[0:?] OF apm_domain_instance;
   copies_of_key_values_before_linking : LIST[0:?] OF apm_primitive_domain_instance;
END_ENTITY;

ENTITY apm_object_domain_instance
  SUBTYPE OF( apm_complex_domain_instance );
   domain : apm_object_domain;
END_ENTITY;

ENTITY apm_multi_level_domain_instance
  SUBTYPE OF( apm_complex_domain_instance );
   domain : apm_multi_level_domain;
END_ENTITY;

ENTITY apm_primitive_domain_instance
   ABSTRACT SUPERTYPE OF (ONEOF(apm_real_instance , apm_string_instance ) )
   SUBTYPE OF( apm_domain_instance );
   domain : apm_primitive_domain;
   has_value : BOOLEAN;
   is_input : BOOLEAN;
END_ENTITY;

ENTITY apm_real_instance
   SUBTYPE OF( apm_primitive_domain_instance );
   value : REAL;
END_ENTITY;

ENTITY apm_string_instance
   SUBTYPE OF( apm_primitive_domain_instance );
   value : STRING;
END_ENTITY;


(* APM Relations Definitions *)

ENTITY apm_relation
   ABSTRACT SUPERTYPE OF (ONEOF( apm_product_relation ,
                                 apm_product_idealization_relation ) );
   relation_name : STRING;
   relation : STRING;
   related_attributes : LIST[1:?] OF STRING;
END_ENTITY;

ENTITY apm_product_relation
   SUBTYPE OF( apm_relation );
END_ENTITY;

ENTITY apm_product_idealization_relation
   SUBTYPE OF( apm_relation );
END_ENTITY;
```

```
(* APM Source Data Wrapper Definitions *)

ENTITY apm_source_data_wrapper_returned_value
   ABSTRACT SUPERTYPE OF (ONEOF( apm_source_data_wrapper_returned_object ,
                                 apm_source_data_wrapper_returned_null_value ,
                                 apm_source_data_wrapper_returned_real_value ,
                                 apm_source_data_wrapper_returned_string_value ,
                                 apm_source_data_wrapper_returned_list ) );
   name : STRING;
END_ENTITY;

ENTITY apm_source_data_wrapper_object;
   data_file_name : STRING;
END_ENTITY;

ENTITY apm_source_data_wrapper_returned_object
   SUBTYPE OF( apm_source_data_wrapper_returned_value );
   values : LIST[1:?] OF apm_source_data_wrapper_returned_value;
END_ENTITY;

ENTITY apm_source_data_wrapper_returned_null_value
   SUBTYPE OF( apm_source_data_wrapper_returned_value );
END_ENTITY;

ENTITY apm_source_data_wrapper_returned_real_value
   SUBTYPE OF( apm_source_data_wrapper_returned_value );
   value : REAL;
END_ENTITY;

ENTITY apm_source_data_wrapper_returned_string_value
   SUBTYPE OF( apm_source_data_wrapper_returned_value );
   value : STRING;
END_ENTITY;

ENTITY apm_source_data_wrapper_returned_list
   SUBTYPE OF( apm_source_data_wrapper_returned_value );
   elements : LIST[1:?] OF apm_source_data_wrapper_returned_value;
END_ENTITY;


(* APM Solver Definitions *)

ENTITY apm_solver_wrapper;
END_ENTITY;

ENTITY mathematica_wrapper
   SUBTYPE OF( apm_solver_wrapper );
END_ENTITY;

ENTITY apm_solver_result;
   results : LIST[1:?] OF REAL;
END_ENTITY;
```

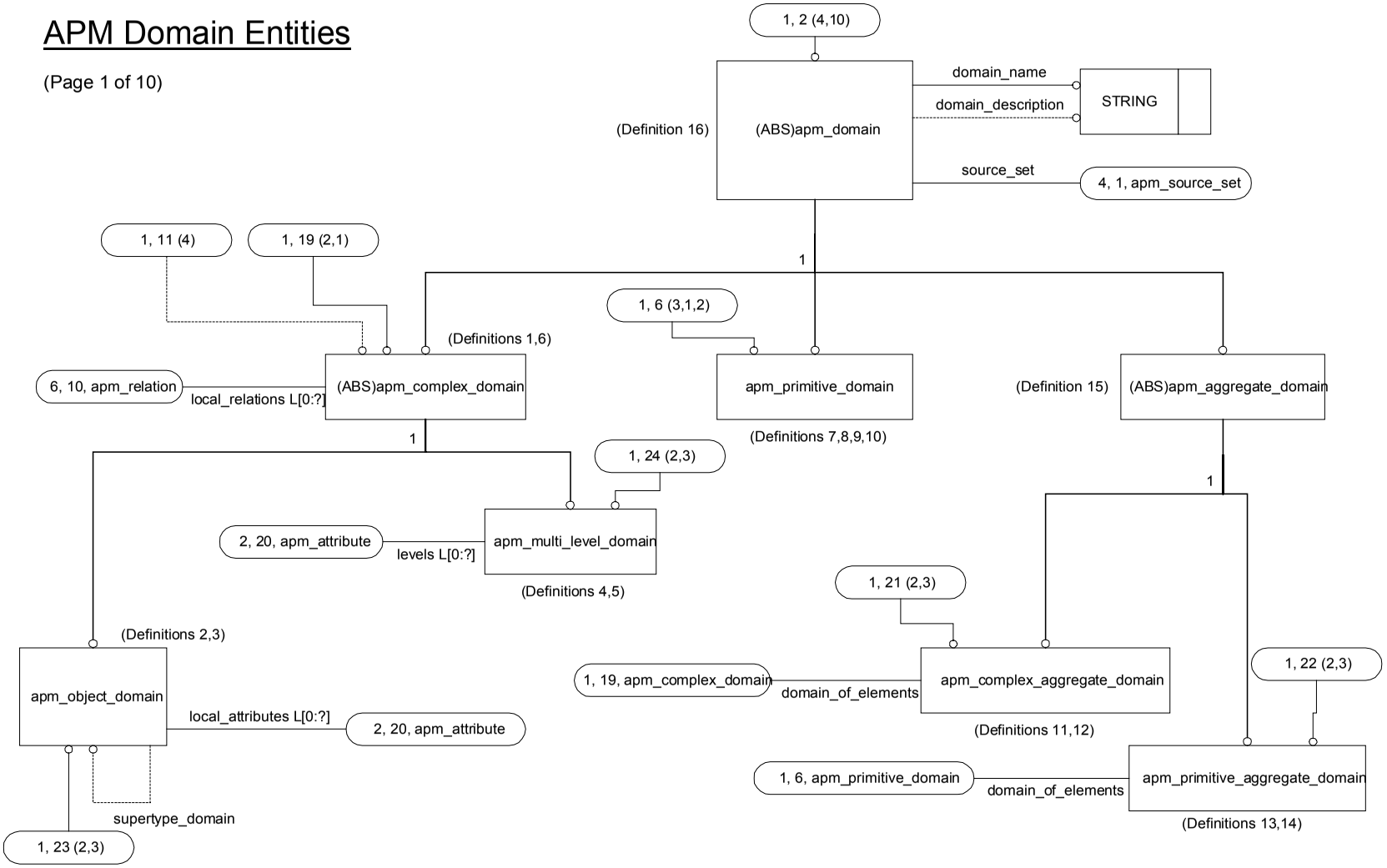
```
(* Constraint Network Definitions *)

ENTITY constraint_network;
   relations : LIST[0:?] OF constraint_network_relation;
   variables : LIST[0:?] OF constraint_network_variable;
END_ENTITY;

ENTITY constraint_network_node
   ABSTRACT SUPERTYPE OF (ONEOF(constraint_network_relation ,
                                constraint_network_variable ) );
   name : STRING;
   marked : BOOLEAN;
   constraint_network : constraint_network;
END_ENTITY;

ENTITY constraint_network_relation
   SUBTYPE OF( constraint_network_node );
   expression : STRING;
   variables : LIST[0:?] OF constraint_network_variable;
   active : BOOLEAN;
   category : INTEGER;
END_ENTITY;

ENTITY constraint_network_variable
   SUBTYPE OF( constraint_network_node );
   relations : LIST[0:?] OF constraint_network_relation;
END_ENTITY;

END_SCHEMA;
```

APPENDIX F

APM INFORMATION MODEL EXPRESS-G DIAGRAMS

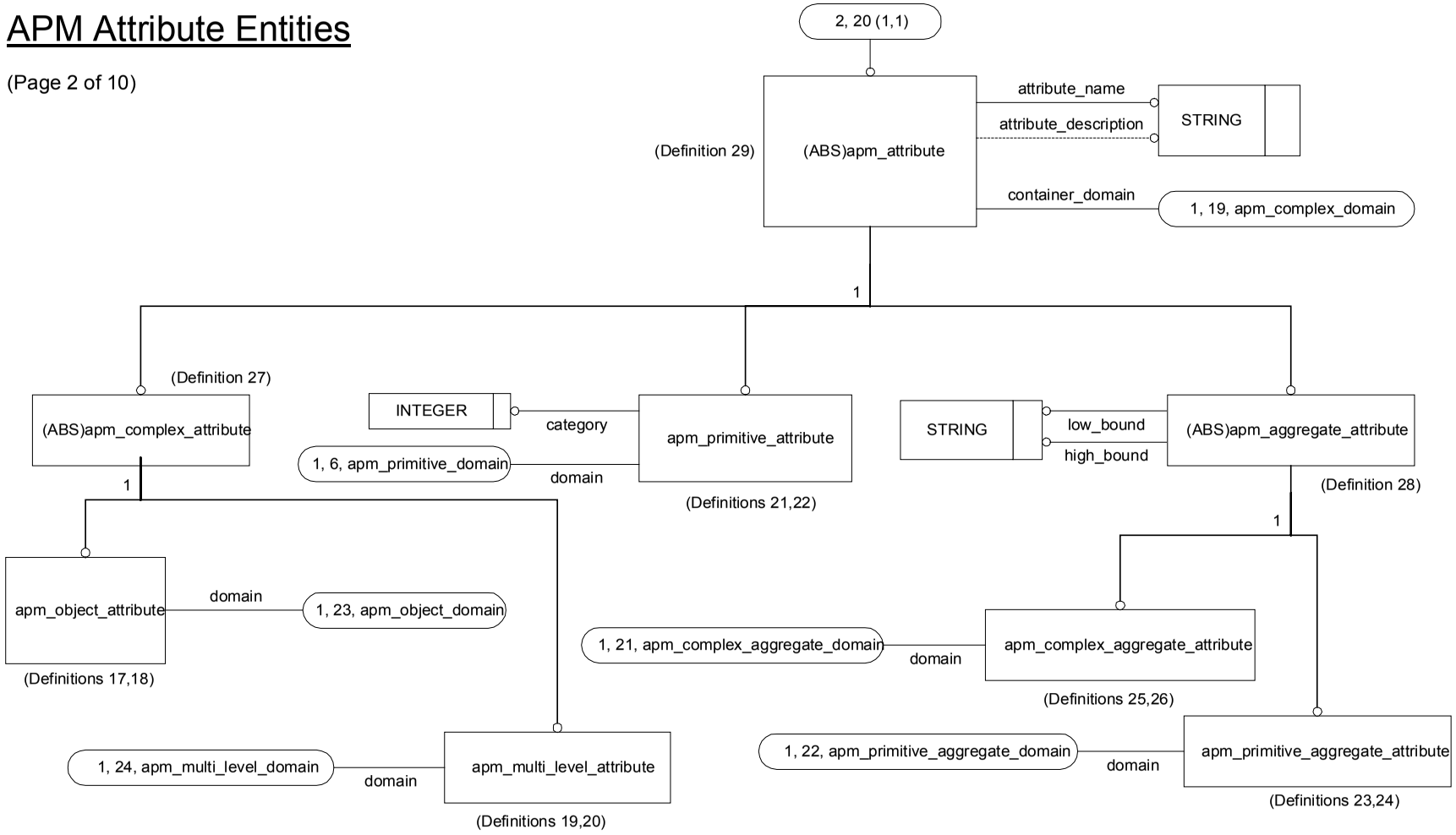
APM Domain Entities

(Page 1 of 10)



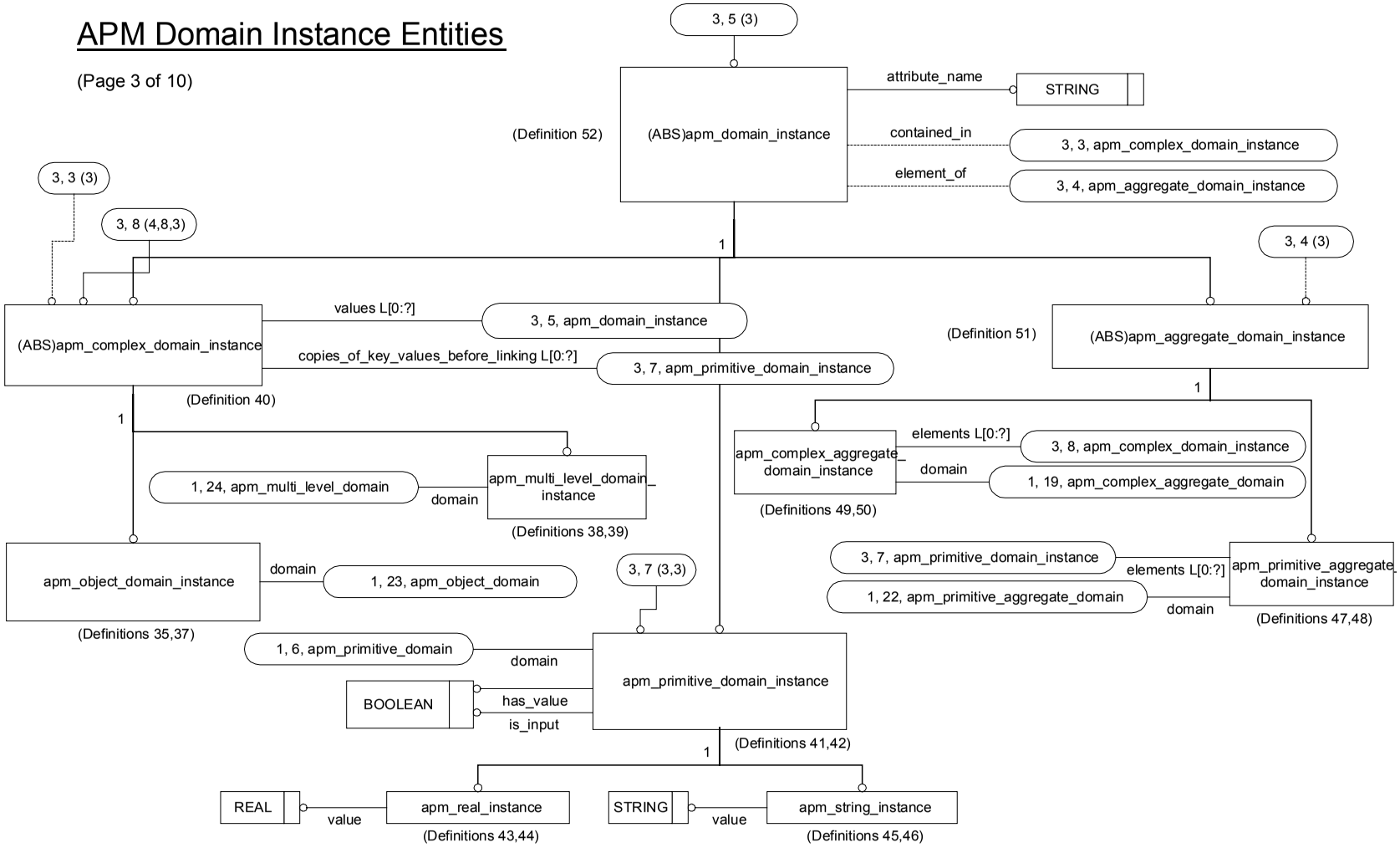
APM Attribute Entities

(Page 2 of 10)



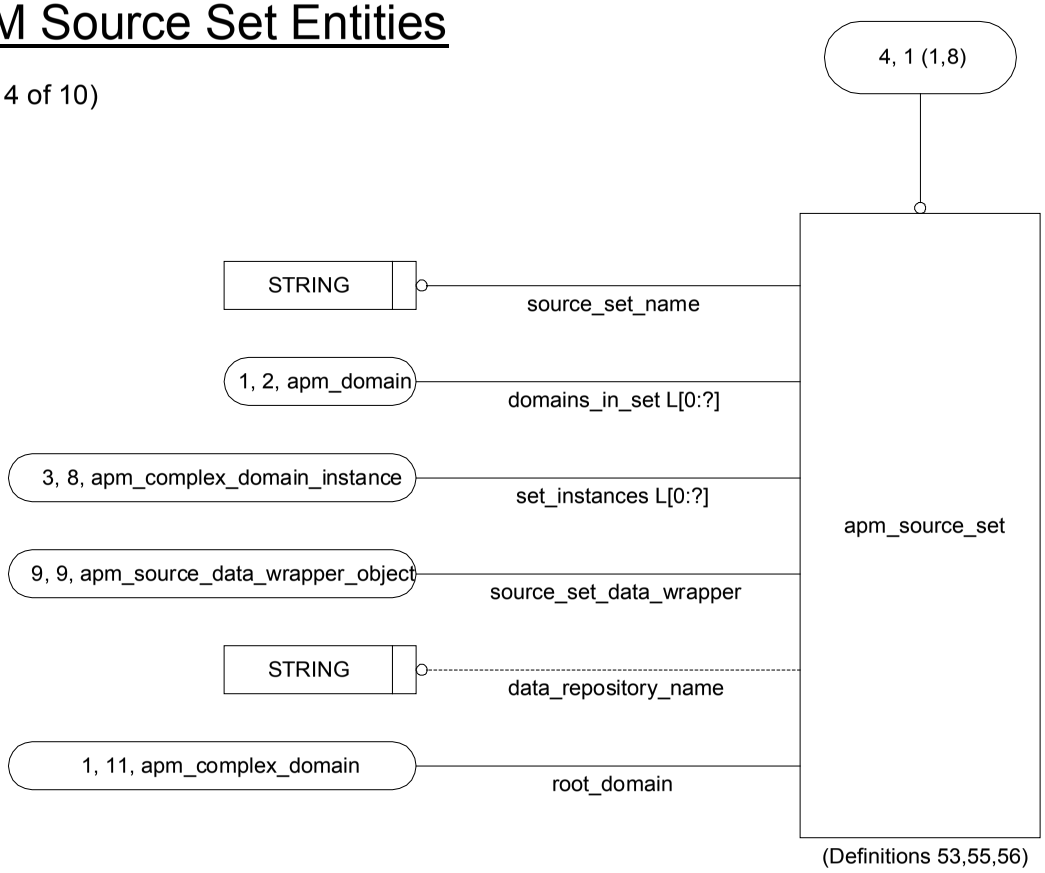
APM Domain Instance Entities

(Page 3 of 10)



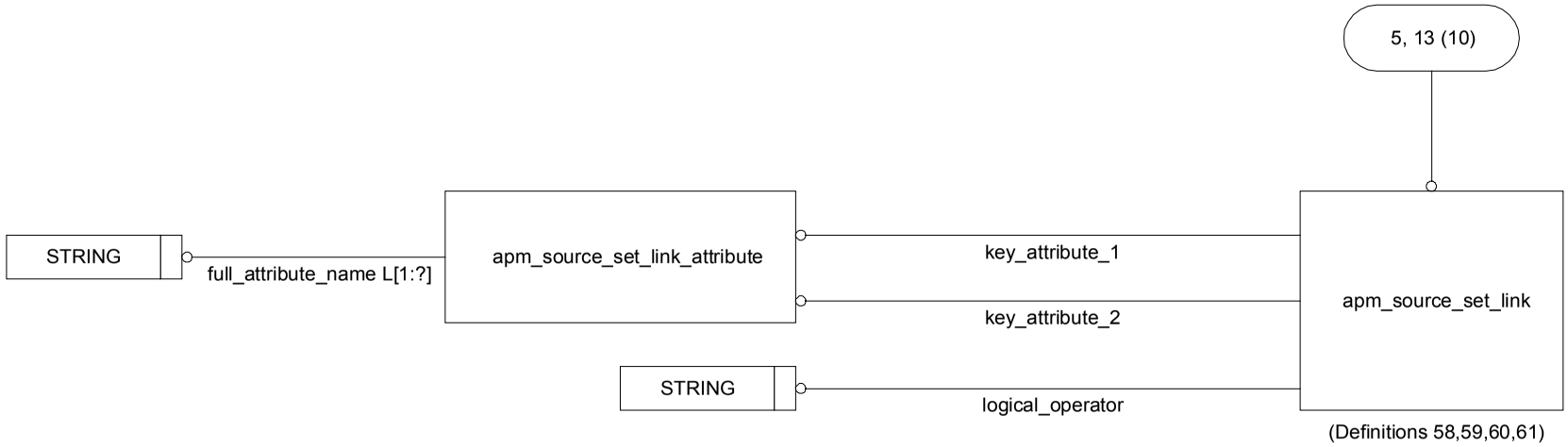
APM Source Set Entities

(Page 4 of 10)



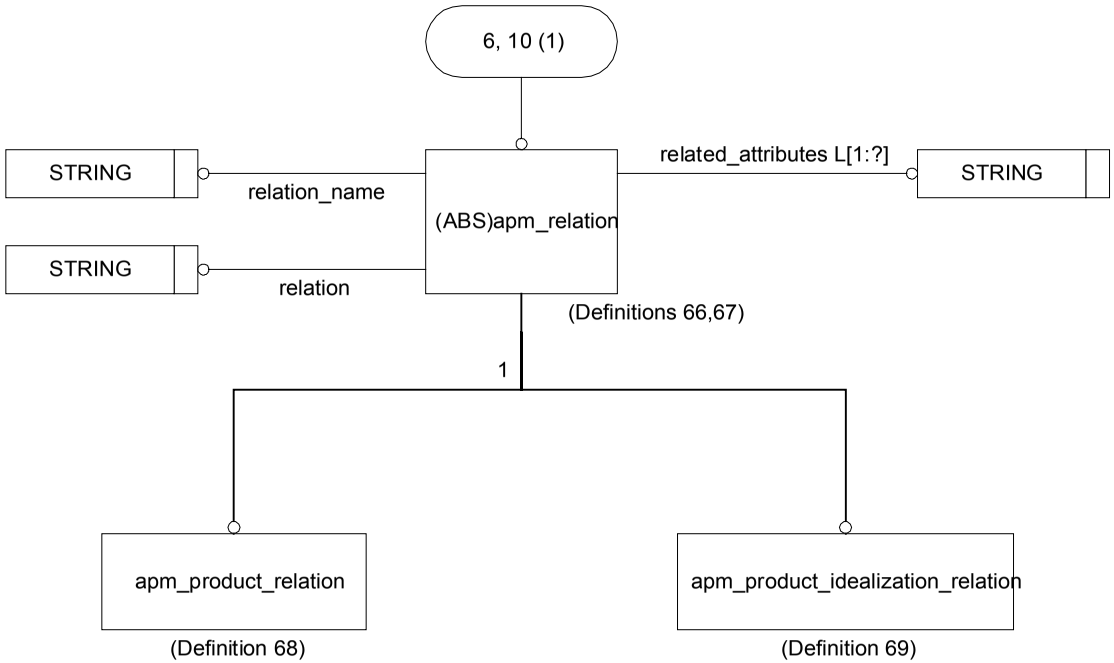
APM Source Set Link Entities

(Page 5 of 10)



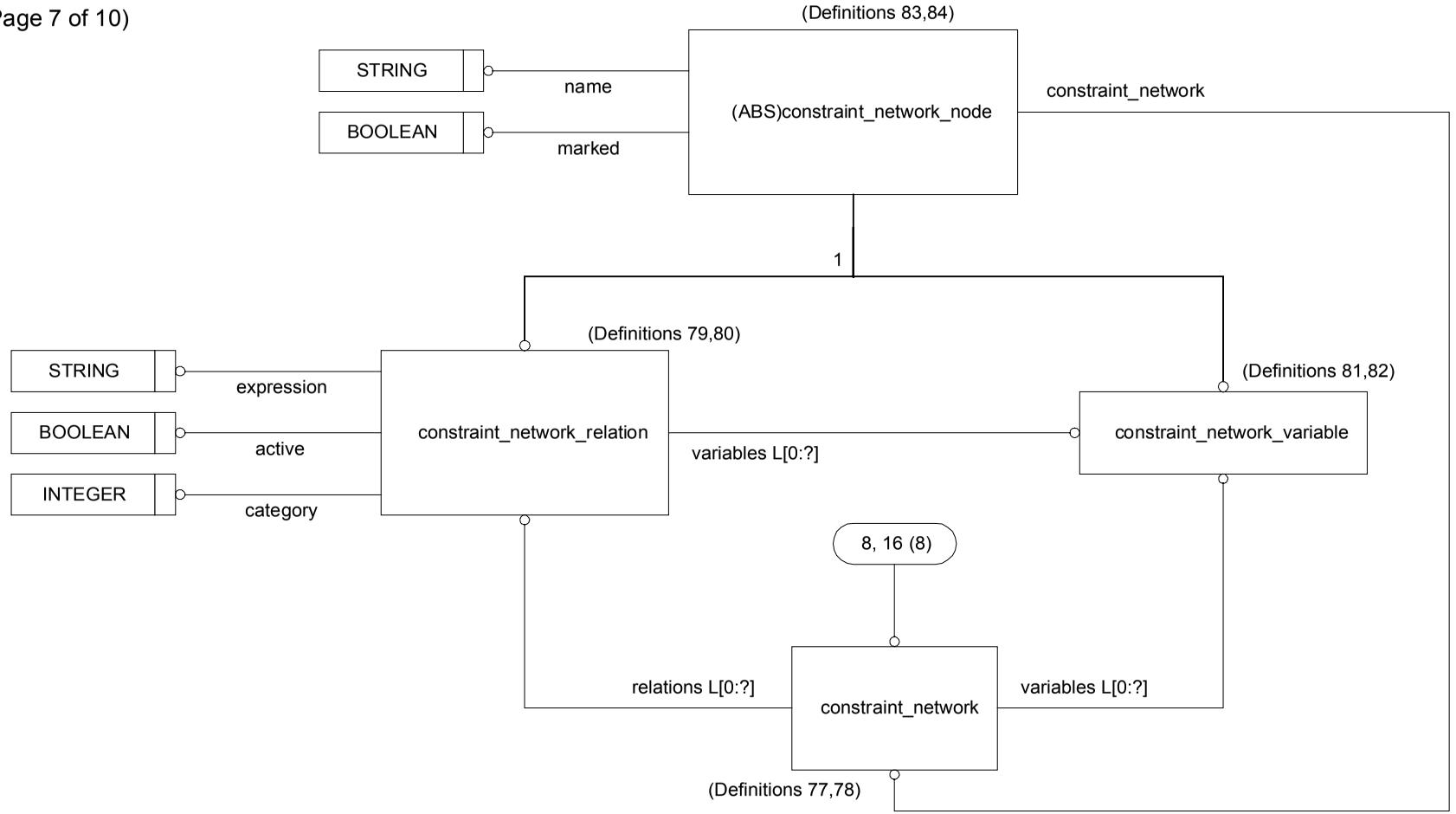
APM Relation Entities

(Page 6 of 10)



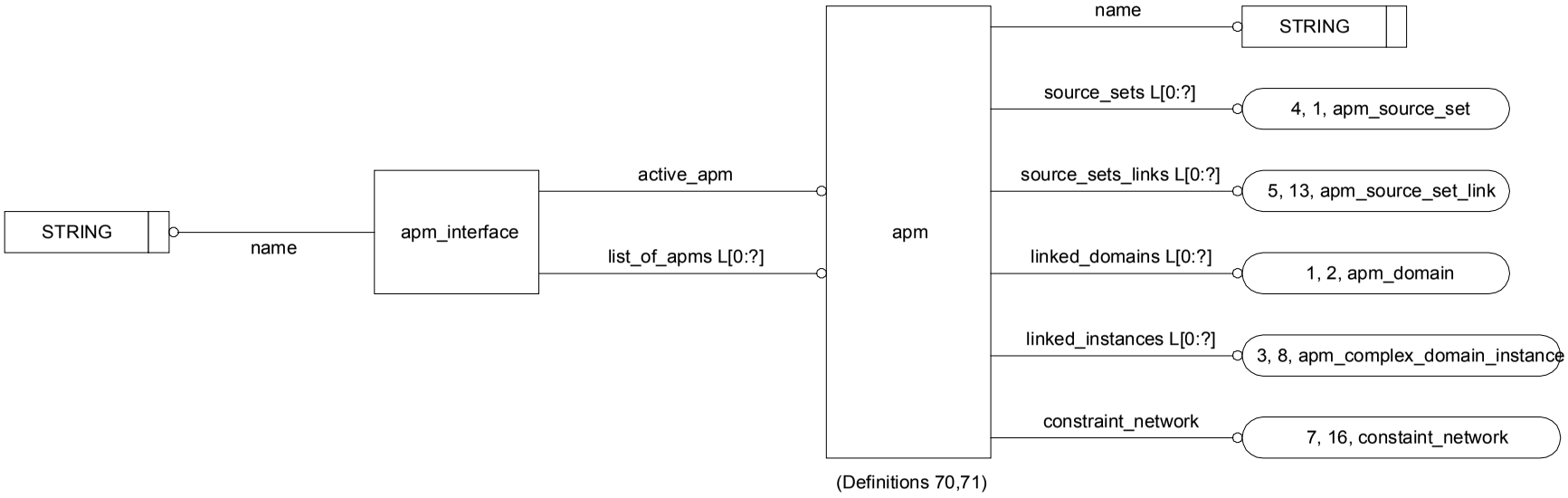
Constraint Network Entities

(Page 7 of 10)



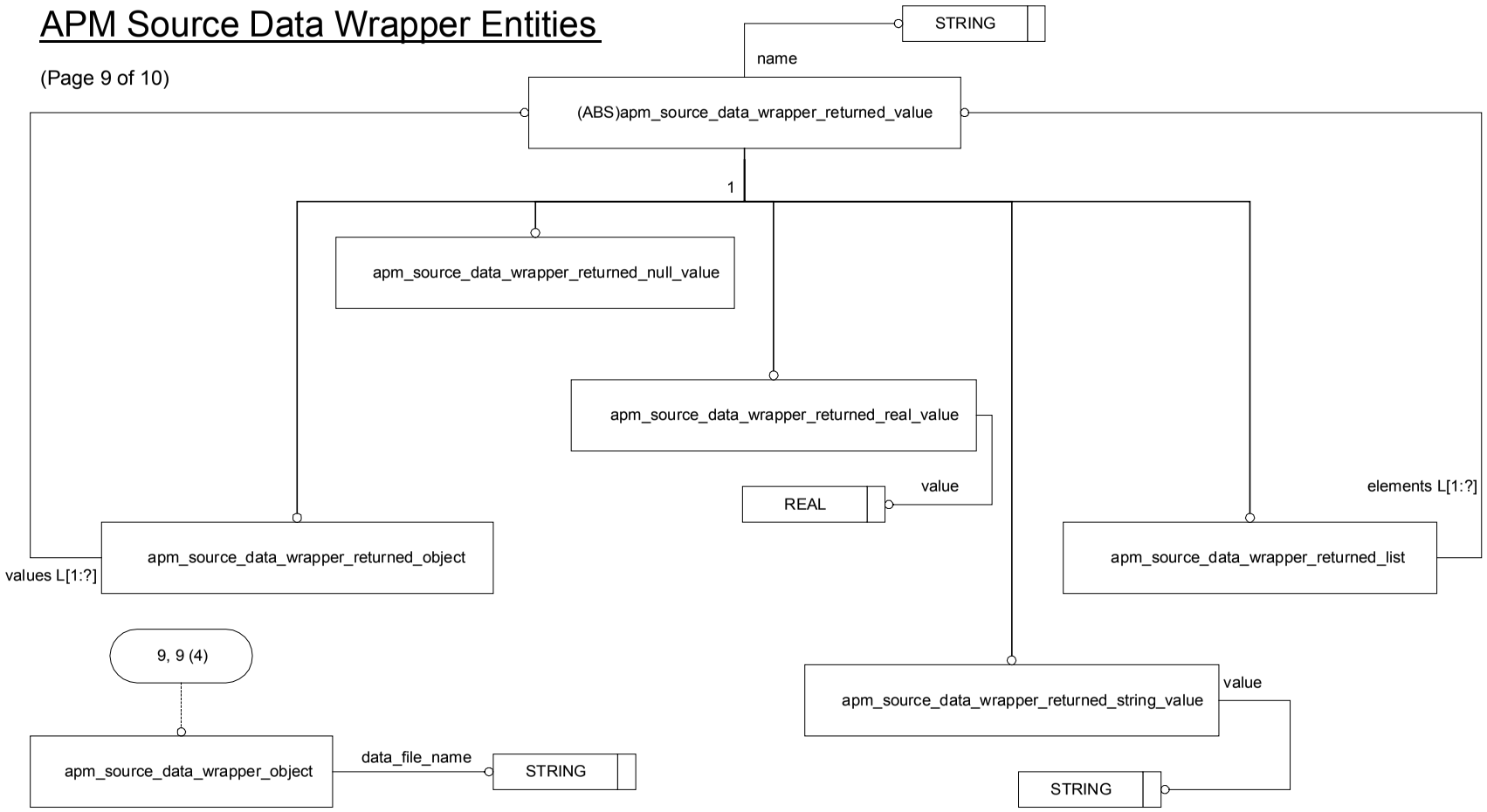
APM Entities

(Page 8 of 10)



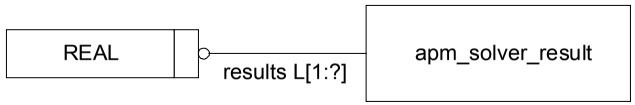
APM Source Data Wrapper Entities

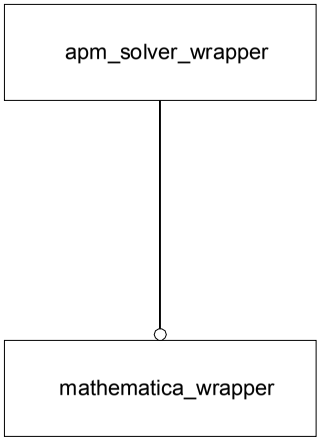
(Page 9 of 10)



APM Solver Wrapper Entities

(Page 10 of 10)





APPENDIX G

PROTOTYPE CLASS IMPLEMENTATIONS

L.1 Class APMInterface Prototype Implementation

```
package apm;
import constraint.*;


public class APMInterface
{
    private static String name;
    private static APM activeAPM;
    private static ListOfAPMs listOfAPMs;

    // Constructor: name only
    public APMInterface( String n )
    {
        name = n;
        listOfAPMs = new ListOfAPMs();
    }


    // Initialization Methods
    public static void initialize()
    {
        listOfAPMs = new ListOfAPMs();
    }


    // APM Interface Methods
    public static boolean loadAPMDefinitions( String fileName )
    {
        boolean success = false;

        APM apmInstance = new APM( name );
        success = apmInstance.loadAPMDefinitions( fileName );

        // Add the loaded apm to the list of apms
        listOfAPMs.addElement( apmInstance );

        // Make it active
        activeAPM = apmInstance;

        return success;
    }

    public static ListOfAPMSourceSets getSourceSets()
    {
        return activeAPM.getSourceSets();
    }

    public static boolean loadSourceSetData( ListOfStrings listOfFileNames )
    {
        return activeAPM.loadSourceSetData( listOfFileNames );
    }

    public static void saveLinkedAPMDefinition( String fileName )
    {
        activeAPM.saveLinkedAPMDefinition( fileName );
    }

    public static void saveInstancesBySourceSet( ListOfStrings outputFileNames )
    {
        activeAPM.saveInstancesBySourceSet( outputFileNames );
    }


    public static void saveLinkedInstances( String outputFileName )
    {
        activeAPM.saveLinkedInstances( outputFileName );
    }
```

```
public static void exportToExpress( String directoryName )
{
    activeAPM.exportToExpress( directoryName );
}

public static String printLinkedAPMDefinitions( )
{
    return activeAPM.printLinkedAPMDefinitions( );
}

public static String printUnlinkedAPMDefinitions( )
{
    return activeAPM.printUnlinkedAPMDefinitions( );
}

public static void printLinkedAPMDefinitions( String outputFileName )
{
    activeAPM.printLinkedAPMDefinitions( outputFileName );
}

public static void printUnlinkedAPMDefinitions( String outputFileName )
{
    activeAPM.printUnlinkedAPMDefinitions( outputFileName );
}

public static String printLinkedAPMInstances( )
{
    return activeAPM.printLinkedAPMInstances( );
}

public static String printUnlinkedAPMInstances( )
{
    return activeAPM.printUnlinkedAPMInstances( );
}


public static void printLinkedAPMInstances( String outputFileName )
{
    activeAPM.printLinkedAPMInstances( outputFileName );
}

public static void printUnlinkedAPMInstances( ListOfStrings outputFileNames )
{
    activeAPM.printUnlinkedAPMInstances( outputFileNames );
}


public static ConstraintNetwork getConstraintNetwork()
{
    return activeAPM.getConstraintNetwork();
}

public static APMDomain getAPMDomain( String sourceSetName , String domainName )
{
    return activeAPM.getAPMDomain( sourceSetName , domainName );
}

public static ListOfAPMComplexDomainInstances getInstancesOf( String domainName )
{
    return activeAPM.getInstancesOf( domainName );
}


// Get Methods
public APM getActiveAPM()
{
    return activeAPM;
```

```
}

public ListOfAPMs getListOfAPMs()
{
    return listOfAPMs;
}

public APM getAPM( String name )
{
    for( int i = 0 ; i < listOfAPMs.size() ; i++ )
        if( listOfAPMs.elementAt( i ).getName().equals( name ) )
            return listOfAPMs.elementAt( i );

    return null;
}


// Set Methods
public void setActiveAPM( APM apm )
{
    // If the apm is in the list of apms, just make it the active one
    if( this.isInAPMList( apm ) )
        activeAPM = apm;

    // If it is not in the list, add it to the lsit and make it the active one
    else
    {
        this.addAPM( apm );
        activeAPM = apm;
    }


}

public void setActiveAPM( String name )
{
    APM newAPM;

    // If the apm is in the list, just make it the active one
    if( this.isInAPMList( name ) )
        activeAPM = this.getAPM( name );

    // If it is not in the list, create an empty one and make it the active
    else
    {
        newAPM = new APM( name );
        this.addAPM( newAPM );
        activeAPM = newAPM;
    }

}


public void addAPM( APM apm )
{
    listOfAPMs.addElement( apm );
}



// Interrogation Methods
public boolean isInAPMList( APM apm )
{
    for( int i = 0 ; i < listOfAPMs.size() ; i++ )
        if( listOfAPMs.elementAt( i ) == apm )
            return true;

    return false;
```

```
    }

    public boolean isInAPMList( String name )
    {
        for( int i = 0 ; i < listOfAPMs.size() ; i++ )
            if( listOfAPMs.elementAt( i ).getName().equals( name ) )
                return true;

        return false;
    }
}
```

L.2 Class APM Prototype Implementation

```
package apm;
import apm.wrapper.*;
import constraint.*;
import java_cup.runtime.Symbol;
import apm.parser.*;
import java.util.*;
import java.io.*;

import java.lang.*;

// NOTE: Most of the methods of this class are package access. They are intended to be called
// through APMInterface and not directly.

public class APM extends java.lang.Object
{
    private String name;
    private ListOfAPMSourceSets sourceSets;
    private ListOfAPMSourceSetLinks sourceSetLinks;
    private ListOfAPMComplexDomainInstances linkedInstances;
    private ListOfAPMDomains linkedDomains;
    private BufferedReader wrapperRegistryFile;
    private BufferedWriter wrapperFactoryFile;
    private BufferedWriter outputFile;
    private static ConstraintNetwork constraintNetwork;

    private static boolean definitionsLoaded = false;


    // Constructor: name only
    public APM( String n )
    {
        name = n;
        sourceSets = new ListOfAPMSourceSets();
        sourceSetLinks = new ListOfAPMSourceSetLinks();
        linkedInstances = new ListOfAPMComplexDomainInstances();
        linkedDomains = new ListOfAPMDomains();
        constraintNetwork = new ConstraintNetwork();
    }

    // Constructor: name and source sets
    public APM( String n , ListOfAPMSourceSets sets )
    {
        name = n;
        sourceSets = sets;
        sourceSetLinks = new ListOfAPMSourceSetLinks();
        linkedInstances = new ListOfAPMComplexDomainInstances();
        linkedDomains = new ListOfAPMDomains();
        constraintNetwork = new ConstraintNetwork();
    }

    // Constructor: name, source sets and source set links
    public APM( String n , ListOfAPMSourceSets sets , ListOfAPMSourceSetLinks links )
    {
        name = n;
        sourceSets = sets;
        sourceSetLinks = links.createCopy();
        linkedInstances = new ListOfAPMComplexDomainInstances();
        linkedDomains = new ListOfAPMDomains();
        constraintNetwork = new ConstraintNetwork();
    }


    // Get methods
    String getName()
    {
        return name;
    }
```

```
ListOfAPMSourceSets getSourceSets()
{
    return sourceSets;
}

ListOfAPMDomains getLinkedDomains()
{
    return linkedDomains;
}

APMSourceSet getSourceSet( String name )
{
    for( int i = 0 ; i < sourceSets.size() ; i++ )
        if( sourceSets.elementAt( i ).getSourceSetName().equals( name ) )
            return sourceSets.elementAt( i );

    return null;
}

APMDomain getAPMDomain( String sourceSetName , String domainName )
{
    // Returns an APMComplexDomain with the name domainName in the list of domains stored
    // in linkedDomains that also belongs to the set with name sourceSetName (to avoid
    // possible duplicate names in different source sets)

    APMDomain tempAPMDomain;

    for( int i = 0 ; i < linkedDomains.size() ; i++ )
    {
        tempAPMDomain = linkedDomains.elementAt( i );
        if( tempAPMDomain.getDomainName().equals( domainName ) &&
            tempAPMDomain.getSourceSet().getSourceSetName().equals( sourceSetName ))
        return tempAPMDomain;
    }


    return null;
}



ListOfAPMSourceSetLinks getSourceSetLinks()
{
    return sourceSetLinks;
}


ListOfAPMComplexDomainInstances getLinkedInstances()
{
    return linkedInstances;
}

static ConstraintNetwork getConstraintNetwork()
{
    return constraintNetwork;
}



ListOfAPMComplexDomainInstances getInstancesOf( String sourceSetName , String domainName )
{
    // Gets the instances of domainName (and its subtypes) from *linkedInstances*
    // whose source set is sourceSetName.
    // Useful when it is possible to have domains with the same name in
    // two different source sets.
```

```
    ListOfAPMComplexDomainInstances resultList = new ListOfAPMComplexDomainInstances();

    // Check each instance in the linkedInstances list of this interface
    for( int i = 0 ; i < linkedInstances.size() ; i++ )
        if(  (  (APMComplexDomain)  linkedInstances.elementAt(  i  ).getDomain()  ).getSourceSet().getSourceSetName().equals(
    sourceSetName ) &&
            ( (APMComplexDomain) linkedInstances.elementAt( i ).getDomain() ).isSubtypeOf( domainName ) )
            resultList.addElement( linkedInstances.elementAt( i ) );

    return resultList;
}

ListOfAPMComplexDomainInstances getInstancesOf( String domainName )
{
    // Get the instances of domainName (and its subtypes) WITHOUT checking the source set
    // to which the domain belongs (there is another version of this method that does)
    // Should only be used when sure that there are no conflicts with the domainName


    ListOfAPMComplexDomainInstances resultList = new ListOfAPMComplexDomainInstances();

    // Check each instance in the linkedInstances list of this interface
    for( int i = 0 ; i < linkedInstances.size() ; i++ )
        if( ( (APMComplexDomain) linkedInstances.elementAt( i ).getDomain() ).isSubtypeOf( domainName ) )
            resultList.addElement( linkedInstances.elementAt( i ) );

    return resultList;
}



// Add methods

void addSourceSet( APMSourceSet s )
{
    sourceSets.addElement( s );
}

void addSourceSetLink( APMSourceSetLink l )
{
    sourceSetLinks.addElement( l );
}

void addLinkedInstance( APMComplexDomainInstance i )
{
    linkedInstances.addElement( i );
}

void addLinkedInstances( ListOfAPMComplexDomainInstances list )
{
    for( int i = 0 ; i < list.size() ; i++ )
        this.addLinkedInstance( list.elementAt( i ) );
}

void addLinkedDomains( ListOfAPMDomains doms )
{
    for( int i = 0 ; i < doms.size() ; i++ )
        linkedDomains.addElement( doms.elementAt( i ) );

}

void addRelationsToConstraintNetwork( String nameSuffix , APMComplexDomain d )
{
    APMRelation tempRelation;
    ListOfAPMAttributes listOfAttributes = new ListOfAPMAttributes();
    APMAttribute tempAttribute;
    APMComplexDomain domainOfElements;
```

```
    APMSourceSet tempAttributeSourceSet;
    ListOfAPMComplexDomains subtypesOfDomainOfElements;
    APMComplexDomain tempSubtype;

    // If d has relations, add them to the constraint network
    if( d.getRelations().size() != 0 )
        for( int i = 0 ; i < d.getRelations().size() ; i++ )
        {
            tempRelation = d.getRelations().elementAt( i );
            if( tempRelation.isAnAPMProductRelation() )
                this.constraintNetwork.addRelation( new ConstraintNetworkRelation( nameSuffix , tempRelation.getRelationName() ,
                    constraintNetwork , tempRelation.getRelation() , APMRelation.PRODUCT_RELATION ) );
            else if( tempRelation.isAnAPMProductIdealizationRelation() )
                this.constraintNetwork.addRelation( new ConstraintNetworkRelation( nameSuffix , tempRelation.getRelationName() ,
                    constraintNetwork , tempRelation.getRelation() , APMRelation.PRODUCT_IDEALIZATION_RELATION ) );
        }


    // Get the list of attributes depending on whether d is object or multi-level
    if( d.isAnAPMObjectDomain() )
        listOfAttributes = ( (APMObjectDomain) d ).getAttributes();
    else if( d.isAnAPMMultiLevelDomain() )
        listOfAttributes = ( (APMMultiLevelDomain) d ).getLevels();

    // Recursively call this method for each complex attribute of d
    for( int i = 0 ; i < listOfAttributes.size() ; i++ )
    {
        tempAttribute = listOfAttributes.elementAt( i );

        if( tempAttribute.getDomain().isAnAPMComplexDomain() )
            this.addRelationsToConstraintNetwork( nameSuffix + "." + tempAttribute.getAttributeName() , (APMComplexDomain)
                tempAttribute.getDomain() );

        if( tempAttribute.getDomain().isAnAPMComplexAggregateDomain() )
        {
            // Instances of tempAttribute may be of any subtype of domainOfElements
            // Therefore, we have to add a relation to the constraint network
            // for each possible form (domain) the elements may take.
            domainOfElements = (APMComplexDomain) ( (APMComplexAggregateDomain) tempAttribute.
                getDomain() ).getDomainOfElements();
            tempAttributeSourceSet = tempAttribute.getDomain().getSourceSet();
            subtypesOfDomainOfElements = tempAttributeSourceSet.getSubtypesOf( domainOfElements );

            for( int j = 0 ; j < subtypesOfDomainOfElements.size() ; j++ )
            {
                tempSubtype = subtypesOfDomainOfElements.elementAt( j );

                // Add the relation: note that the name passed includes the domain name.
                // (according to the convention, the full name of an attribute inside
                // a complex aggregate includes the domain name to be able to distinguish
                // among the different subtypes the element may be)
                this.addRelationsToConstraintNetwork( nameSuffix + "." + tempAttribute.getAttributeName() + "." +
                    tempSubtype.getDomainName(), tempSubtype );
            }
        }
    }
}


// Data loading and linking methods
boolean loadAPMDefinitions( String fileName )
{
    APMParser theAPMParser = new APMParser( fileName );
    Symbol symbolReturnedFromParser = null;
    APM tempAPM = new APM( "dummy name" );
    APM dummyInterface = new APM( "Dummy Interface" );
```

```
    System.out.println( "Loading APM definitions from file \"" + fileName + "\"...\n" );

    try
    {
        symbolReturnedFromParser = theAPMParser.parse();
    }
    catch ( Exception e )
    {
        System.err.println( "Problems parsing definition file \"" + fileName + "\"" );
        System.err.println( "Definitions not read" );
        return false;
    }

    // Get the APM from the symbol returned from the parser
    tempAPM = (APM) symbolReturnedFromParser.value;

    // Get the name of the interface parsed
    this.name = tempAPM.getName();

    // Get the source set of the interface parsed
    this.sourceSets = tempAPM.getSourceSets();

    // Get the source set links of the interface parsed
    this.sourceSetLinks = tempAPM.getSourceSetLinks();


    // Make a flat COPY of these definitions in the linkedDomains attribute of this interface
    // To obtain a copy, we reload the APM definitions into a dummy interface

    if(!definitionsLoaded ) // This is to avoid infinite recursion!
    {
        definitionsLoaded = true;
        dummyInterface.loadAPMDefinitions( fileName );

        for( int i = 0 ; i < dummyInterface.sourceSets.size() ; i++ )
            this.addLinkedDomains( dummyInterface.getSourceSets().elementAt( i ).getDomainsInSet() );

        // Link the APM Definitions
        this.linkAPMDefinitions();


        // Build the constraint network
        this.createConstraintNetwork();

        definitionsLoaded = false; // Set it back to false in case we want to read a different APM later


    }


    // Definitions read OK
    return true;

}

boolean loadSourceSetData( ListOfStrings listOfFileNames )
{
    APMSourceSet tempSourceSet;
    APMSourceDataWrapperObject wrapperObject = null;
    String tempFileName;
    APMComplexDomain tempSourceSetRootDomain;
    ListOfAPMSourceDataWrapperReturnedObjects returnedListOfObjects = new ListOfAPMSourceDataWrapperReturnedObjects();
    APMComplexDomainInstance instance = null;
    APMComplexDomainInstance instanceCopy = null;
    APMSourceDataWrapperReturnedObject tempReturnedObject;
    ListOfAPMComplexDomains listOfSourceSetRootDomainSubtypes = new ListOfAPMComplexDomains();
    APMComplexDomain tempSourceSetRootDomainSubtype;
    boolean objectsLoaded = false;
```

```
System.out.println( "\nLoading source set data: " );


// Load the data for each source set
for( int i = 0 ; i < this.getSourceSets().size() ; i++ )
{
    // Get the name of the data file for this set from the argument list
    tempFileName = listOfFileNames.elementAt( i ); // The file were the data is

    // Get a source set
    tempSourceSet = this.getSourceSets().elementAt( i );

    // Ask the APMSourceDataWrapperFactory to create a WraperObject for this set
    wrapperObject = APMSourceDataWrapperFactory.makeWrapperObjectFor( tempSourceSet.getSourceSetName( ) ,
tempFileName );

    // Set the wrapperObject attribute of this source set to the wrapperObject just created
    tempSourceSet.setSourceSetDataWrapper( wrapperObject );

    // Get the root domain of this source set
    tempSourceSetRootDomain = tempSourceSet.getRootDomain();

    // Get a list of all domains in this subset that are subtyped (directly or
    // indirectly) from the root domain (the list will include the root domain as
    // a subtype of itself) (we want to read instances of all the subtypes of the
    // root domain, not only of the root domain)
    // If the root domain is a multi-level domain it will not, by definition
    // have subtypes.
    listOfSourceSetRootDomainSubtypes = tempSourceSet.getSubtypesOf( tempSourceSetRootDomain );

    // For each subtype of the root domain, load the instances of it that exist in the
    // data file
    for( int j = 0 ; j < listOfSourceSetRootDomainSubtypes.size() ; j++ )
    {
        tempSourceSetRootDomainSubtype = listOfSourceSetRootDomainSubtypes.elementAt( j );

        // Ask the wrapper to get the values of these attributes
        returnedListOfObjects = wrapperObject.getInstancesOf( tempSourceSetRootDomainSubtype );

        // Check if any objects were returned
        if( returnedListOfObjects.size() > 0 )
            objectsLoaded = true;


        // For each returned object, create a complex domain instance with the values returned
        for( int k = 0 ; k < returnedListOfObjects.size() ; k++ )
        {
            // Get a returned object from the list
            tempReturnedObject = returnedListOfObjects.elementAt( k );

            // If the root domain is an APMObjectDomain, create an APMObjectDomainInstance
            if( tempSourceSetRootDomainSubtype.isAnAPMObjectDomain() )
            {
                instance = new APMObjectDomainInstance( "root" , (APMObjectDomain) tempSourceSetRootDomainSubtype );
                instanceCopy = new APMObjectDomainInstance( "root" , (APMObjectDomain)
                tempSourceSetRootDomainSubtype );
            }

            // If the root domain is an APMMultiLevelDomain, create an APMMultiLevelDomainInstance
            else if( tempSourceSetRootDomainSubtype.isAnAPMMultiLevelDomain() )
            {
                instance = new APMMultiLevelDomainInstance( "root" , (APMMultiLevelDomain)
            tempSourceSetRootDomainSubtype );
                instanceCopy = new APMMultiLevelDomainInstance( "root" , (APMMultiLevelDomain)
            tempSourceSetRootDomainSubtype );
            }
```

```
                // Fill the instance with the values for this object received from the WrapperObject
                instance.populateWithValues( tempReturnedObject.getValues() );
                instanceCopy.populateWithValues( tempReturnedObject.getValues() );


                // Add this instance to the list of instances of the set
                tempSourceSet.getSetInstances().addElement( instance );

                // Add a COPY of the instances of this set to the list of apm linkedInstances
                // This is the list of linkedInstances that gets linked later
                this.getLinkedInstances().addElement( instanceCopy );
            }
        }
    }

    System.out.println( "Done loading source set data." );
    System.out.flush();


    // If any objects were loaded, proceed to link the data
    if( objectsLoaded )
        this.linkSourceSetData();

    // Return whether or not any objects were loaded
    return objectsLoaded;
}


void linkAPMDefinitions()
{
    APMSourceSetLink tempSourceSetLink;
    String sourceSetName1;
    String sourceSetName2;
    String domainName1;
    String domainName2;
    APMComplexDomain domain1;
    APMComplexDomain domain2;
    ListOfStrings key1;
    ListOfStrings key2;
    String keyLeaf1;
    String keyLeaf2;
    ListOfStrings relativeKey1 = new ListOfStrings();
    ListOfStrings relativeKey2 = new ListOfStrings();
    APMAttribute insertionAttribute;
    APMAttribute insertedAttribute;
    APMComplexDomain insertionNode;
    APMObjectDomain objectDomainToBeUpdated;
    int indexOfAttributeToBeUpdated;
    APMComplexDomain transplantedTreeRootNode;

    System.out.println( "Linking APM definitions..." );

    for( int i = 0 ; i < sourceSetLinks.size() ; i++ )
    {
        // Get a source set link
        tempSourceSetLink = sourceSetLinks.elementAt( i );


        // Get the full names of the key attributes
        key1 = tempSourceSetLink.getKeyAttribute1().getFullAttributeName();
        key2 = tempSourceSetLink.getKeyAttribute2().getFullAttributeName();


        // Get the source set names and the domain names
        sourceSetName1 = key1.elementAt( 0 );
        domainName1 = key1.elementAt( 1 );
        sourceSetName2 = key2.elementAt( 0 );
```

```
    domainName2 = key2.elementAt( 1 );

    // Get both domains
    domain1 = (APMComplexDomain) this.getAPMDomain( sourceSetName1 , domainName1 );
    domain2 = (APMComplexDomain) this.getAPMDomain( sourceSetName2 , domainName2 );


    // Discard the first two names from the full names (which correspond to
    // the source set name and the domain name, respectively )
    for( int j = 2 ; j < key1.size() ; j++ )
        relativeKey1.addElement( key1.elementAt( j ) );
    for( int j = 2 ; j < key2.size() ; j++ )
        relativeKey2.addElement( key2.elementAt( j ) ) ;

    // Get the last name in the key name list
    keyLeaf1 = key1.elementAt( key1.size() - 1 );
    keyLeaf2 = key2.elementAt( key2.size() - 1 );

    // Get the key attributes
    insertionAttribute = domain1.getAttribute( relativeKey1 );
    insertedAttribute = domain2.getAttribute( relativeKey2 );

    // Clear lists for next loop
    relativeKey1 = new ListOfStrings();
    relativeKey2 = new ListOfStrings();

    // Get the domains of the insertion node and the transplanted tree root
    insertionNode = insertionAttribute.getContainerDomain();
    transplantedTreeRootNode = insertedAttribute.getContainerDomain();

    // If the insertion node is an APMObjectDomain, we could potentially be
    // modifying one of its supertypes instead of the domain of the node itself.
    if( insertionNode.isAnAPMObjectDomain() )
    {
        // Get the supertype that contains the key attribute
        objectDomainToBeUpdated = ( (APMObjectDomain ) insertionNode ).getSupertypeContainingAttribute( keyLeaf1 );
        indexOfAttributeToBeUpdated = objectDomainToBeUpdated.getIndexOfLocalAttribute( keyLeaf1 );

        // Assign a new domain to the insertion node.
        if( transplantedTreeRootNode.isAnAPMObjectDomain() )
            objectDomainToBeUpdated.getLocalAttributes( ).setElementAt( new APMObjectAttribute( keyLeaf1 ,
                objectDomainToBeUpdated , (APMObjectDomain) transplantedTreeRootNode ) , indexOfAttributeToBeUpdated );
        else if( transplantedTreeRootNode.isAnAPMMultiLevelDomain() )
            objectDomainToBeUpdated.getLocalAttributes( ).setElementAt( new APMMultiLevelAttribute( keyLeaf1 ,
                objectDomainToBeUpdated , (APMMultiLevelDomain) transplantedTreeRootNode ) ,
                indexOfAttributeToBeUpdated );
    }
    else if ( insertionNode.isAnAPMMultiLevelDomain() )
    {
        indexOfAttributeToBeUpdated = ((APMMultiLevelDomain) insertionNode ).getIndexOfLevel( keyLeaf1 );

        // Assign a new domain to the insertion node.
        if( transplantedTreeRootNode.isAnAPMObjectDomain() )
            ((APMMultiLevelDomain) insertionNode ).getLevels( ).setElementAt( new APMObjectAttribute( keyLeaf1 ,
                (APMComplexDomain) insertionNode , (APMObjectDomain) transplantedTreeRootNode ) ,
                indexOfAttributeToBeUpdated );
        else if( transplantedTreeRootNode.isAnAPMMultiLevelDomain() )
            ((APMMultiLevelDomain) insertionNode ).getLevels( ).setElementAt( new APMMultiLevelAttribute( keyLeaf1 ,
                (APMComplexDomain)insertionNode , (APMMultiLevelDomain) transplantedTreeRootNode ) ,
                indexOfAttributeToBeUpdated );
    }

}


System.out.println( "Done Linking APM definitions..." );
System.out.flush();
```

```
}

void linkSourceSetData()
{
    APMSourceSetLink tempSourceSetLink;
    String sourceSetName1;
    String sourceSetName2;
    String domainName1;
    String domainName2;
    ListOfStrings fullKeyAttributeName1;
    ListOfStrings fullKeyAttributeName2;
    String lastKeyAttributeName1;
    String lastKeyAttributeName2;
    ListOfStrings keyAttributeName1 = new ListOfStrings();
    ListOfStrings keyAttributeName2 = new ListOfStrings();
    APMComplexDomain domain1 = null;
    APMComplexDomain domain2 = null;
    ListOfAPMComplexDomainInstances listOfDomain1Instances;
    ListOfAPMComplexDomainInstances listOfDomain2Instances;
    APMComplexDomainInstance tempDomain1Instance;
    APMComplexDomainInstance tempDomain2Instance;
    APMPrimitiveDomainInstance key1Instance = null;
    APMPrimitiveDomainInstance key1InstanceCopy = null;
    APMPrimitiveDomainInstance key2Instance = null;
    ListOfAPMPrimitiveDomainInstances listOfKey1Instances;
    ListOfAPMPrimitiveDomainInstances listOfKey2Instances;
    APMComplexDomainInstance containingInstance1 = null;
    APMComplexDomainInstance containingInstance2 = null;
    APMComplexDomainInstance containingInstance2Copy = null;
    int attributeIndex1;
    int attributeIndex2;
    APMComplexDomainInstance emptyDomain2Instance =null;
    boolean matchFound = false;



    System.out.println( "Linking source set data..." );

    for( int i = 0 ; i < sourceSetLinks.size() ; i++ )
    {
        // Get a source set link definition
        tempSourceSetLink = sourceSetLinks.elementAt( i );

        // Get a few names needed
        fullKeyAttributeName1 = tempSourceSetLink.getKeyAttribute1().getFullAttributeName();
        fullKeyAttributeName2 = tempSourceSetLink.getKeyAttribute2().getFullAttributeName();
        sourceSetName1 = fullKeyAttributeName1.elementAt( 0 );
        sourceSetName2 = fullKeyAttributeName2.elementAt( 0 );
        domainName1 = fullKeyAttributeName1.elementAt( 1 );
        domainName2 = fullKeyAttributeName2.elementAt( 1 );
        lastKeyAttributeName1 = fullKeyAttributeName1.elementAt( fullKeyAttributeName1.size() -1 );
        lastKeyAttributeName2 = fullKeyAttributeName2.elementAt( fullKeyAttributeName2.size() -1 );

        // Extract the key attribute names from the full names (i.e., cut the first
        // two names, which correspond to the source set name and the domain name)
        keyAttributeName1 = new ListOfStrings();
        keyAttributeName2 = new ListOfStrings();
        for( int j = 2 ; j < fullKeyAttributeName1.size() ; j++ )
            keyAttributeName1.addElement( fullKeyAttributeName1.elementAt( j ) );
        for( int j = 2 ; j < fullKeyAttributeName2.size() ; j++ )
            keyAttributeName2.addElement( fullKeyAttributeName2.elementAt( j ) );

        // Get domain1 and domain2 (to be used later if there are no matches)
        domain1 = (APMComplexDomain) this.getSourceSet( sourceSetName1 ).getDomain( domainName1 );
        domain2 = (APMComplexDomain) this.getSourceSet( sourceSetName2 ).getDomain( domainName2 );

        // Get all the root instances of domain 1
```

```
listOfDomain1Instances = getInstancesOf( sourceSetName1 , domainName1 );

// Get all the root instances of domain 2
listOfDomain2Instances = getInstancesOf( sourceSetName2 , domainName2 );


// For each instance of domain 1
for( int cursor1 = 0 ; cursor1 < listOfDomain1Instances.size() ; cursor1++ )
{
    // Get an instance of domain1 from the list
    tempDomain1Instance = listOfDomain1Instances.elementAt( cursor1 );

    // Get the instances of key attribute 1 in tempDomain1Instance
    // (there may be more than one if the key attribute is in an aggregate)
    listOfKey1Instances = (ListOfAPMPrimitiveDomainInstances)tempDomain1Instance.getInstances( keyAttributeName1 );

    // For each key instance in the domain1 instance
    for( int m = 0 ; m < listOfKey1Instances.size() ; m++ )
    {
        matchFound = false;

        // Get a key 1 instance
        key1Instance = listOfKey1Instances.elementAt( m );


        // For each instance of domain 2
        for( int cursor2 = 0 ; cursor2 < listOfDomain2Instances.size() ; cursor2++ )
        {
            if( matchFound )
                break;

            // Get an instance of domain2 from the list
            tempDomain2Instance = listOfDomain2Instances.elementAt( cursor2 );

            // Get the instances of key attribute 2 in tempDomain2Instance
            // (there may be more than one if the key attribute is in an aggregate)
            listOfKey2Instances = (ListOfAPMPrimitiveDomainInstances) tempDomain2Instance.getInstances(
        keyAttributeName2 );

            // For each key instance in the domain1 instance

            for( int n = 0 ; n < listOfKey2Instances.size() ; n++ )
            {
                // Get a key 2 instance
                key2Instance = listOfKey2Instances.elementAt( n );


                // Compare key1Instance and key2Instance
                if( ( key1Instance.isAnAPMStringInstance() && ( ( (APMStringInstance)
                    key1Instance ).getStringValue().equals( ( (APMStringInstance) key2Instance ).getStringValue() ) ) ) ||
                    ( key1Instance.isAnAPMRealInstance() && ( ( (APMRealInstance) key1Instance ).getRealValue() == (
                    (APMRealInstance) key2Instance ).getRealValue() ) ) )
                {
                    // key1Instance and key2Instance match



                    // Get the container of key1Instance
                    containingInstance1 = key1Instance.getContainedIn();

                    // Store a copy of key1Instance in containingInstance1's
                    // copiesOfKeyValuesBeforeLinking attribute
                    key1InstanceCopy = (APMPrimitiveDomainInstance) key1Instance.createCopy();
                    containingInstance1.addCopyOfKeyValueBeforeLinking( key1InstanceCopy );

                    // Get the index of key1Instance in this container
                    attributeIndex1 = containingInstance1.getIndexOf( lastKeyAttributeName1 );
```

```
                        // Get the container of key2Instance
                        containingInstance2 = key2Instance.getContainedIn();

                        // Create a COPY of containingInstance2
                        containingInstance2Copy = (APMComplexDomainInstance) containingInstance2.createCopy();

                        // Change the attribute name of this copy
                        containingInstance2Copy.setAttributeName( lastKeyAttributeName1 );

                        // Point attribute of container1Instance in position attributeIndex1 to
                        // the copy of containing2Instance just created
                        containingInstance1.getValues().setElementAt( containingInstance2Copy , attributeIndex1 );

                        // Set the containedIn attribute of containingInstance2Copy
                        containingInstance2Copy.setContainedIn( containingInstance1 );


                        // Set the matchFound flag to true
                        matchFound = true;


                        break;
                    }

                }

            }
            if( ! matchFound )
            {
                // If we have gotten this far is because no match for key1Instance was found
                // Assign an empty instance of domain2 in the place where key2Instance should have been

                containingInstance1 = key1Instance.getContainedIn();
                attributeIndex1 = containingInstance1.getIndexOf( lastKeyAttributeName1 );

                if( domain2.isAnAPMObjectDomain() )
                    emptyDomain2Instance = new APMObjectDomainInstance( lastKeyAttributeName1 ,
                        containingInstance1 , (APMObjectDomain) domain2 );
                else if( domain2.isAnAPMMultiLevelDomain() )
                    emptyDomain2Instance = new APMMultiLevelDomainInstance( lastKeyAttributeName1 ,
                        containingInstance1 , (APMMultiLevelDomain) domain2 );

                emptyDomain2Instance.instantiateAllAttributes();

                // Change the attribute name so that it matches the new (linked) attribute name
                key1Instance.setAttributeName( lastKeyAttributeName2 );

                containingInstance1.getValues().setElementAt( emptyDomain2Instance , attributeIndex1 );

            }


        }



    }
}

System.out.println( "Done linking source set data." );
System.out.flush();
}

// Information display methods

String printUnlinkedAPMDefinitions( )
{
```

```
    StringBuffer s = new StringBuffer();
    APMSourceSet tempAPMSourceSet;
    APMSourceSetLink tempAPMSourceSetLink;
    String keyAttributeName1 , keyAttributeName2;
    String logicalOperator;

    // Print source sets
    for( int i = 0 ; i < this.getSourceSets().size() ; i++ )
    {
        tempAPMSourceSet = this.getSourceSets().elementAt( i );
        s.append( "==================================================\n" );
        s.append( "Source set: \"" + tempAPMSourceSet.getSourceSetName() + "\"\n" );
        s.append( "==================================================\n\n" );

        s.append( printAPMDomains( tempAPMSourceSet.getDomainsInSet() ) );

        s.append( "\n" );
    }


    // Print source set links
    s.append( "==================================================\n" );
    s.append( "Source set links: \n" );
    s.append( "==================================================\n\n" );

    for( int i = 0 ; i < this.getSourceSetLinks().size() ; i++ )
    {
        tempAPMSourceSetLink = this.getSourceSetLinks().elementAt( i );
        keyAttributeName1 = tempAPMSourceSetLink.getKeyAttribute1().printFullAttributeName();
        keyAttributeName2 = tempAPMSourceSetLink.getKeyAttribute2().printFullAttributeName();
        logicalOperator = tempAPMSourceSetLink.getLogicalOperator();
        s.append( "LINK[ " + i + "] : " + keyAttributeName1 + " " + logicalOperator + " " + keyAttributeName2 + "\n" );

    }


    return s.toString();
}


void printUnlinkedAPMDefinitions( String outputFileName )
{
    StringBuffer body = new StringBuffer();

    try
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }

    body.append( this.printUnlinkedAPMDefinitions() );


    // Write the file's body
    try
    {
        outputFile.write( body.toString() , 0 , body.toString().length() );
    }

    catch( IOException e )
    {
        System.err.println( "Error writing to file\n" + e.toString() );
        System.exit( 1 );
```

```
    }

    try
    {
        outputFile.flush();
        outputFile.close();
    }
    catch ( IOException e )
    {
        System.err.println( "Error closing file\n" + e.toString() );
        System.exit( 1 );
    }


}


String printLinkedAPMDefinitions()
{
    StringBuffer s = new StringBuffer();
    s.append( printAPMDomains( this.getLinkedDomains() ) );
    return s.toString();
}


void printLinkedAPMDefinitions( String outputFileName )
{
    StringBuffer body = new StringBuffer();

    body.append( printAPMDomains( this.getLinkedDomains() ) );

    // Create and open the output file for writing
    try
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }


    // Write the file's body
    try
    {
        outputFile.write( body.toString() , 0 , body.toString().length() );
    }

    catch( IOException e )
    {
        System.err.println( "Error writing to file\n" + e.toString() );
        System.exit( 1 );
    }

    try
    {
        outputFile.flush();
        outputFile.close();
    }
    catch ( IOException e )
    {
        System.err.println( "Error closing file\n" + e.toString() );
        System.exit( 1 );
    }
```

```
}


String printSourceSets()
{
    StringBuffer s = new StringBuffer();
    APMSourceSet tempAPMSourceSet;
    APMDomain tempAPMDomain;


    for( int sourceSetCounter = 0 ; sourceSetCounter < this.getSourceSets().size() ; sourceSetCounter++ )
    {
        tempAPMSourceSet = this.getSourceSets().elementAt( sourceSetCounter );
        s.append( "==================================================\n" );
        s.append( "Source set: \"" + tempAPMSourceSet.getSourceSetName() + "\"\n" );
        s.append( "==================================================\n\n" );

        s.append( printAPMDomains( tempAPMSourceSet.getDomainsInSet() ) );

        s.append( "\n" );
    }




    return s.toString();
}


String printAPMDomains( ListOfAPMDomains listOfAPMDomains )
{
    APMDomain tempAPMDomain;
    StringBuffer s = new StringBuffer();

    for( int i = 0 ; i < listOfAPMDomains.size() ; i++ )
    {
        tempAPMDomain = listOfAPMDomains.elementAt( i );
        if( ! tempAPMDomain.isAnAPMPrimitiveDomain() && !tempAPMDomain.isAnAPMPrimitiveAggregateDomain() ) // Don't
        print primitive domains
            s.append( tempAPMDomain.toString() + "\n" );
    }

    return s.toString();

}

String printUnlinkedAPMInstances( )
{
    StringBuffer s = new StringBuffer();
    APMSourceSet tempAPMSourceSet;
    APMComplexDomainInstance tempAPMComplexDomainInstance;

    for( int i = 0 ; i < this.getSourceSets().size() ; i++ )
    {
        tempAPMSourceSet = this.getSourceSets().elementAt( i );
        s.append( "==================================================\n" );
        s.append( "Source set: \"" + tempAPMSourceSet.getSourceSetName() + "\"\n" );
        s.append( "==================================================\n\n" );

        for( int j = 0 ; j < tempAPMSourceSet.getSetInstances().size() ; j++ )
        {
            tempAPMComplexDomainInstance = tempAPMSourceSet.getSetInstances().elementAt( j );
            if( tempAPMComplexDomainInstance.getAttributeName().equals( "root" ) )
                s.append( tempAPMComplexDomainInstance.toString() );
        }
    }
```

```
    return s.toString();



}
String printLinkedAPMInstances()
{
    StringBuffer s = new StringBuffer();

    // Display only instances of a subtype of the root domain of
    // the first source set
    String rootDomainName = this.getSourceSets().elementAt( 0 ).getRootDomain().getDomainName();

    for( int i = 0 ; i < linkedInstances.size() ; i++ )
        if( ( (APMComplexDomain) linkedInstances.elementAt( i ).getDomain() ).isSubtypeOf( rootDomainName ) )
            s.append( linkedInstances.elementAt( i ).toString() );


    return s.toString();
}



// Save Methods
void saveLinkedAPMDefinition( String outputFileName )
{
    StringBuffer body = new StringBuffer();
    APMDomain tempDomain;
    ListOfAPMAttributes listOfAttributes = new ListOfAPMAttributes();
    ListOfAPMRelations listOfRelations = new ListOfAPMRelations();
    APMObjectDomain tempObjectDomain;
    APMMultiLevelDomain tempMultiLevelDomain;
    APMAttribute tempAttribute;
    APMRelation tempRelation;
    String lowBound;
    String highBound;
    boolean tempDomainHasLocalProductRelations;
    boolean tempDomainHasLocalProductIdealizationRelations;



    try
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }


    // Create the body of the file

    body.append( "APM " + name + ";\n\n" );


    body.append( "SOURCE_SET unified_apm " + "ROOT_DOMAIN " +
        this.sourceSets.elementAt( 0 ).getRootDomain().getDomainName() + ";\n\n" );

    // Write each domain
    for( int j = 0 ; j < this.linkedDomains.size() ; j++ )
    {
        tempDomain = linkedDomains.elementAt( j );
        tempDomainHasLocalProductRelations = false;
        tempDomainHasLocalProductIdealizationRelations = false;
```

```
// Don't write primitive domains
if( tempDomain.isAnAPMPrimitiveDomain() || tempDomain.isAnAPMPrimitiveAggregateDomain() )
    continue;

// Write the appropriate header for APMObjectDomains
if( tempDomain.isAnAPMObjectDomain() )
{
    // Cast it
    tempObjectDomain = (APMObjectDomain) tempDomain;

    body.append( "DOMAIN " + tempObjectDomain.getDomainName() );
    if( tempObjectDomain.hasSupertype() )
        body.append( " SUBTYPE_OF " + tempObjectDomain.getSupertypeDomain().getDomainName() );
        body.append( ";\n" );

    listOfAttributes = tempObjectDomain.getLocalAttributes();
    listOfRelations = tempObjectDomain.getLocalRelations();
    tempDomainHasLocalProductRelations = tempObjectDomain.hasLocalProductRelations();
    tempDomainHasLocalProductIdealizationRelations = tempObjectDomain.hasLocalProductIdealizationRelations();

}
// Write the appropriate header for APMMultiLevelDomains
else if( tempDomain.isAnAPMMultiLevelDomain() )
{
    // Cast it
    tempMultiLevelDomain = (APMMultiLevelDomain) tempDomain;
    body.append( "MULTI_LEVEL_DOMAIN " + tempMultiLevelDomain.getDomainName() + ";\n" );

    listOfAttributes = tempMultiLevelDomain.getLevels();
    listOfRelations = tempMultiLevelDomain.getLocalRelations();
    tempDomainHasLocalProductRelations = tempMultiLevelDomain.hasLocalProductRelations();
    tempDomainHasLocalProductIdealizationRelations = tempMultiLevelDomain.hasLocalProductIdealizationRelations();
}

// Write the attributes for APMObjectDomains or the levels for APMMultiLevelDomains
// Write the relations for both
if( tempDomain.isAnAPMComplexDomain() )
{
    for( int k = 0 ; k < listOfAttributes.size() ; k++ )
    {
        tempAttribute = listOfAttributes.elementAt( k );

        if( tempAttribute.isAnAPMPrimitiveAttribute() && ( ( (APMPrimitiveAttribute) tempAttribute ).isProductAttribute() ==
    false ) )
            body.append( " IDEALIZED " + tempAttribute.getAttributeName() + " : " +
                tempAttribute.getDomain().getDomainName() + ";\n" );
        else if( tempAttribute.isAnAPMAggregateAttribute() )
        {
            lowBound = ( (APMAggregateAttribute) tempAttribute ).getLowBound();
            highBound = ( (APMAggregateAttribute) tempAttribute ).getHighBound();
            body.append( " " + tempAttribute.getAttributeName() + " : LIST[" + lowBound + "," + highBound + "] OF " +
                 ( (APMAggregateDomain) tempAttribute.getDomain() ).getDomainOfElements().getDomainName() + ";\n" );
        }
        else if( tempAttribute.isAnAPMMultiLevelAttribute() )
            body.append( " " + tempAttribute.getAttributeName() + " : MULTI_LEVEL " +
                tempAttribute.getDomain().getDomainName() + ";\n" );
        else
            body.append( " " + tempAttribute.getAttributeName() + " : " + tempAttribute.getDomain().getDomainName() +
        ";\n" );
    }

    // Write the product idealization relations
    if( tempDomainHasLocalProductIdealizationRelations )
    {
        body.append( "\n PRODUCT_IDEALIZATION_RELATIONS\n" );
        for( int k = 0 ; k < listOfRelations.size() ; k++ )
        {
            tempRelation = listOfRelations.elementAt( k );
```

```
                    if( tempRelation.isAnAPMProductIdealizationRelation() )
                        body.append( " " + tempRelation.getRelationName() + " : \"" + tempRelation.getRelation() + "\";\n" );

                }
            }

            // Write the product relations
            if( tempDomainHasLocalProductRelations )
            {
                body.append( "\n PRODUCT_RELATIONS\n" );
                for( int k = 0 ; k < listOfRelations.size() ; k++ )
                {
                    tempRelation = listOfRelations.elementAt( k );
                    if( tempRelation.isAnAPMProductRelation() )
                        body.append( " " + tempRelation.getRelationName() + " : \"" + tempRelation.getRelation() + "\";\n" );
                }

                body.append( "\n" );
            }


        }

        // Write the appropriate tail for APMObjectDomains
        if( tempDomain.isAnAPMObjectDomain() )
            body.append( "END_DOMAIN;\n\n" );

        // Write the appropriate tail for APMMultiLevelDomains
        else if( tempDomain.isAnAPMMultiLevelDomain() )
        body.append( "END_MULTI_LEVEL_DOMAIN;\n\n" );

    }


    body.append( "END_SOURCE_SET;\n\n" );

    body.append( "END_APM;" );

    try
    {
        outputFile.write( body.toString() , 0 , body.toString().length() );
    }

    catch( IOException e )
    {
        System.err.println( "Error writing to file\n" + e.toString() );
        System.exit( 1 );
    }

    try
    {
        outputFile.flush();
        outputFile.close();
    }
    catch ( IOException e )
    {
        System.err.println( "Error closing file\n" + e.toString() );
        System.exit( 1 );
    }


}

void exportToExpress( String directoryName )
{

    StringBuffer body = new StringBuffer();
    APMSourceSet tempSourceSet;
```

```
String schemaName;
String outputFileName;
ListOfAPMDomains listOfDomains = new ListOfAPMDomains();
APMDomain tempDomain;
ListOfAPMAttributes listOfAttributes = new ListOfAPMAttributes();
ListOfAPMRelations listOfRelations = new ListOfAPMRelations();
APMObjectDomain tempObjectDomain;
APMMultiLevelDomain tempMultiLevelDomain;
APMAttribute tempAttribute;
APMRelation tempRelation;
String lowBound;
String highBound;
boolean tempDomainHasLocalProductRelations;
boolean tempDomainHasLocalProductIdealizationRelations;


for( int i = 0 ; i < this.sourceSets.size() + 1 ; i++ )
{

    if( i < this.sourceSets.size() )
    {
        tempSourceSet = this.sourceSets.elementAt( i );
        schemaName = tempSourceSet.getSourceSetName();
        outputFileName = directoryName + schemaName + ".exp";
        listOfDomains = tempSourceSet.getDomainsInSet();
    }

    else
    {
        schemaName = "unified_apm";
        outputFileName = directoryName + schemaName + ".exp";
        listOfDomains = this.linkedDomains;
    }



    try
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }


    // Create the body of the file
    body = new StringBuffer();
    body.append( "SCHEMA " + schemaName + ";\n\n" );


    // Write each domain
    for( int j = 0 ; j < listOfDomains.size() ; j++ )
    {
        tempDomain = listOfDomains.elementAt( j );
        tempDomainHasLocalProductRelations = false;
        tempDomainHasLocalProductIdealizationRelations = false;

        // Don't write primitive domains
        if( tempDomain.isAnAPMPrimitiveDomain() || tempDomain.isAnAPMPrimitiveAggregateDomain() )
            continue;

        // Write the appropriate header for APMObjectDomains
        if( tempDomain.isAnAPMObjectDomain() )
        {
            // Cast it
```

```
    tempObjectDomain = (APMObjectDomain) tempDomain;

    body.append( "ENTITY " + tempObjectDomain.getDomainName() );
    if( tempObjectDomain.hasSupertype() )
        body.append( " SUBTYPE OF(" + tempObjectDomain.getSupertypeDomain().getDomainName() + ")" );
        body.append( ";\n" );

    listOfAttributes = tempObjectDomain.getLocalAttributes();
    listOfRelations = tempObjectDomain.getLocalRelations();
    tempDomainHasLocalProductRelations = tempObjectDomain.hasLocalProductRelations();
    tempDomainHasLocalProductIdealizationRelations = tempObjectDomain.hasLocalProductIdealizationRelations();

}
// Write the appropriate header for APMMultiLevelDomains
else if( tempDomain.isAnAPMMultiLevelDomain() )
{
    // Cast it
    tempMultiLevelDomain = (APMMultiLevelDomain) tempDomain;
    body.append( "ENTITY " + tempMultiLevelDomain.getDomainName() + ";\n" );

    listOfAttributes = tempMultiLevelDomain.getLevels();
    listOfRelations = tempMultiLevelDomain.getLocalRelations();
    tempDomainHasLocalProductRelations = tempMultiLevelDomain.hasLocalProductRelations();
    tempDomainHasLocalProductIdealizationRelations = tempMultiLevelDomain.hasLocalProductIdealizationRelations();
}

// Write the attributes for APMObjectDomains or the levels for APMMultiLevelDomains
// Write the relations for both
if( tempDomain.isAnAPMComplexDomain() )
{
    for( int k = 0 ; k < listOfAttributes.size() ; k++ )
    {
        tempAttribute = listOfAttributes.elementAt( k );

        if( tempAttribute.isAnAPMPrimitiveAttribute() && ( ( (APMPrimitiveAttribute) tempAttribute ).isIdealizedAttribute()
    ) )
            body.append( " (* IDEALIZED *) " + tempAttribute.getAttributeName() + " : " +
                tempAttribute.getDomain().getDomainName() + ";\n" );
        else if( tempAttribute.isAnAPMPrimitiveAttribute() && ( ( (APMPrimitiveAttribute) tempAttribute
    ).isEssentialAttribute() ) )
            body.append( " (* ESSENTIAL *) " + tempAttribute.getAttributeName() + " : " +
                tempAttribute.getDomain().getDomainName() + ";\n" );

        else if( tempAttribute.isAnAPMAggregateAttribute() )
        {
            lowBound = ( (APMAggregateAttribute) tempAttribute ).getLowBound();
            highBound = ( (APMAggregateAttribute) tempAttribute ).getHighBound();
            body.append( " " + tempAttribute.getAttributeName() + " : LIST[" + lowBound + ":" + highBound + "] OF " +
                ( (APMAggregateDomain) tempAttribute.getDomain() ).getDomainOfElements().getDomainName() + ";\n" );
        }
        else if( tempDomain.isAnAPMMultiLevelDomain() && ( k != 0 ) )
            body.append( " " + tempAttribute.getAttributeName() + " : OPTIONAL " +
                tempAttribute.getDomain().
            getDomainName() + ";\n" );
        else
            body.append( " " + tempAttribute.getAttributeName() + " : " + tempAttribute.getDomain().getDomainName() +
        ";\n" );
    }

    // Write relations as WHERE rules
    // NOTE: Comment WHERE rules for now (because they are not EXPRESS-compliant
    if( tempDomainHasLocalProductIdealizationRelations || tempDomainHasLocalProductRelations )
    {
        body.append( "(* WHERE" );

        // Write Product Idealization Relations
        if( tempDomainHasLocalProductIdealizationRelations )
        {
```

```
                    body.append( "\n (* PRODUCT IDEALIZATION RELATIONS *)\n" );
                    for( int k = 0 ; k < listOfRelations.size() ; k++ )
                    {
                        tempRelation = listOfRelations.elementAt( k );
                        if( tempRelation.isAnAPMProductIdealizationRelation() )
                            body.append( " " + tempRelation.getRelationName() + " : " + tempRelation.getRelation() + ";\n" );

                    }
                }

                // Write the product relations
                if( tempDomainHasLocalProductRelations )
                {
                    body.append( "\n (* PRODUCT RELATIONS *)\n" );
                    for( int k = 0 ; k < listOfRelations.size() ; k++ )
                    {
                        tempRelation = listOfRelations.elementAt( k );
                        if( tempRelation.isAnAPMProductRelation() )
                            body.append( " " + tempRelation.getRelationName() + " : " + tempRelation.getRelation() + ";\n" );
                    }

                    body.append( "\n" );
                }
                body.append( "*)\n" );
            }



        }

        // Write the tail
        if( tempDomain.isAnAPMComplexDomain() )
        body.append( "END_ENTITY;\n\n" );


    }



    body.append( "END_SCHEMA;" );

    try
    {
        outputFile.write( body.toString() , 0 , body.toString().length() );
    }

    catch( IOException e )
    {
        System.err.println( "Error writing to file\n" + e.toString() );
        System.exit( 1 );
    }

    try
    {
        outputFile.flush();
        outputFile.close();
    }
    catch ( IOException e )
    {
        System.err.println( "Error closing file\n" + e.toString() );
        System.exit( 1 );
    }
}


}
```

```
void printUnlinkedAPMInstances( ListOfStrings outputFileNames )
{
    StringBuffer body;
    APMSourceSet tempAPMSourceSet;
    APMComplexDomainInstance tempAPMComplexDomainInstance;

    for( int i = 0 ; i < this.getSourceSets().size() ; i++ )
    {
        try
        {
            outputFile = new BufferedWriter( new FileWriter( outputFileNames.elementAt( i ) ) );
        }

        catch( IOException e )
        {
            System.err.println( "Problems opening file\n" + e.toString() );
            System.exit( 1 );
        }

        // Clear the body text
        body = new StringBuffer();

        tempAPMSourceSet = this.getSourceSets().elementAt( i );
        body.append( "==================================================\n" );
        body.append( "Source set: \"" + tempAPMSourceSet.getSourceSetName() + "\"\n" );
        body.append( "==================================================\n\n" );

        for( int j = 0 ; j < tempAPMSourceSet.getSetInstances().size() ; j++ )
        {
            tempAPMComplexDomainInstance = tempAPMSourceSet.getSetInstances().elementAt( j );
            if( tempAPMComplexDomainInstance.getAttributeName().equals( "root" ) )
                body.append( tempAPMComplexDomainInstance.toString() );
        }

        try
        {
            outputFile.write( body.toString() , 0 , body.toString().length() );
        }

        catch( IOException e )
        {
            System.err.println( "Error writing to file\n" + e.toString() );
            System.exit( 1 );
        }

        try
        {
            outputFile.flush();
            outputFile.close();
        }
        catch ( IOException e )
        {
            System.err.println( "Error closing file\n" + e.toString() );
            System.exit( 1 );
        }

    }

}



void printLinkedAPMInstances( String outputFileName )
{
    StringBuffer body = new StringBuffer();

    // Create and open the output file for writing
    try
```

```
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }


    body.append( printLinkedAPMInstances( ) );


    // Write the file's body
    try
    {
        outputFile.write( body.toString() , 0 , body.toString().length() );
    }

    catch( IOException e )
    {
        System.err.println( "Error writing to file\n" + e.toString() );
        System.exit( 1 );
    }

    try
    {
        outputFile.flush();
        outputFile.close();
    }
    catch ( IOException e )
    {
        System.err.println( "Error closing file\n" + e.toString() );
        System.exit( 1 );
    }


}


void saveInstancesBySourceSet( ListOfStrings outputFileNames )
{
    StringBuffer body = new StringBuffer();
    APMSourceSet tempSourceSet;
    String tempSourceSetName;
    APMComplexDomainInstance tempInstance;


    // Loop through each source set in the APM
    for( int i = 0 ; i < sourceSets.size() ; i++ )
    {
        // Create and open the output file for writing
        try
        {
            outputFile = new BufferedWriter( new FileWriter( outputFileNames.elementAt( i ) ) );
        }

        catch( IOException e )
        {
            System.err.println( "Problems opening file\n" + e.toString() );
            System.exit( 1 );
        }

        // Clear the body text
        body = new StringBuffer();

        body.append( "DATA;\n\n" );
```

```
        // Get a source set
        tempSourceSet = sourceSets.elementAt( i );

        // Get the name of the source set
        tempSourceSetName = tempSourceSet.getSourceSetName();

        // For each instance in the APM, save those that belong to tempSourceSet
        for( int j = 0 ; j < linkedInstances.size() ; j++ )
        {
            tempInstance = linkedInstances.elementAt( j );
            if( tempInstance.getDomain().getSourceSet().getSourceSetName().equals( tempSourceSetName ) )
                body.append( tempInstance.toFlatFormatStringRootInstance( tempSourceSetName ) );
        }

        body.append( "END_DATA;\n" );

        // Write the file's body
        try
        {
            outputFile.write( body.toString() , 0 , body.toString().length() );
        }

        catch( IOException e )
        {
            System.err.println( "Error writing to file\n" + e.toString() );
            System.exit( 1 );
        }

        try
        {
            outputFile.flush();
            outputFile.close();
        }
        catch ( IOException e )
        {
            System.err.println( "Error closing file\n" + e.toString() );
            System.exit( 1 );
        }


    }


}



void saveLinkedInstances( String outputFileName )
{
    StringBuffer body = new StringBuffer();

    // Create and open the output file for writing
    try
    {
        outputFile = new BufferedWriter( new FileWriter( outputFileName ) );
    }

    catch( IOException e )
    {
        System.err.println( "Problems opening file\n" + e.toString() );
        System.exit( 1 );
    }


    // Save only instances of a subtype of the root domain of
    // the first source set
    String rootDomainName = this.getSourceSets().elementAt( 0 ).getRootDomain().getDomainName();
```

```
        body.append( "DATA;\n\n" );

        for( int i = 0 ; i < linkedInstances.size() ; i++ )
            if( ( (APMComplexDomain) linkedInstances.elementAt( i ).getDomain() ).isSubtypeOf( rootDomainName ) )
                body.append( linkedInstances.elementAt( i ).toFlatFormatStringRootInstance() );

        body.append( "END_DATA;\n" );

        // Write the file's body
        try
        {
            outputFile.write( body.toString() , 0 , body.toString().length() );
        }

        catch( IOException e )
        {
            System.err.println( "Error writing to file\n" + e.toString() );
            System.exit( 1 );
        }

        try
        {
            outputFile.flush();
            outputFile.close();
        }
        catch ( IOException e )
        {
            System.err.println( "Error closing file\n" + e.toString() );
            System.exit( 1 );
        }


    }


    void createConstraintNetwork()
    {
        APMDomain tempAPMDomain;

        System.out.println( "\nCreating constraint network..." );

        for( int i = 0 ; i < this.linkedDomains.size() ; i++ )
        {
            tempAPMDomain = this.linkedDomains.elementAt( i );

            if( tempAPMDomain.isAnAPMComplexDomain() )
                addRelationsToConstraintNetwork( tempAPMDomain.getDomainName() , (APMComplexDomain) tempAPMDomain );

        }

        System.out.println( "\nDone creating constraint network.\n" );
        System.out.flush();
    }

}
```

L.3 Class APMRealInstance Prototype Implementation

```
package apm;

import apm.*;
import apm.solver.*;
import constraint.*;
import java.text.DecimalFormat;

public class APMRealInstance extends apm.APMPrimitiveDomainInstance
{
    private double value;



    // Constructor: attribute name and domain
    public APMRealInstance( String n , APMPrimitiveDomain d )
    {
        super( n , d );
        hasValue = false;
    }


    // Constructor: attribute name, domain and value
    public APMRealInstance( String n , APMPrimitiveDomain d , double val )
    {
        super( n , d );
        value = val;
        hasValue = true;

    }

    // Constructor: attribute name, containedIn and domain
    public APMRealInstance( String n , APMComplexDomainInstance c , APMPrimitiveDomain d )
    {
        super( n , c , d );
        hasValue = false;

    }

    // Constructor: attribute name, elementOf and domain
    public APMRealInstance( String n , APMAggregateDomainInstance a , APMPrimitiveDomain d )
    {
        super( n , a , d );
        hasValue = false;

    }

    // Constructor: attribute name, containedIn, domain and value
    public APMRealInstance( String n , APMComplexDomainInstance c , APMPrimitiveDomain d , double val )
    {
    super( n , c , d );
    value = val;
    hasValue = true;

    }

    // Constructor: attribute name, elementOf, domain and value
    public APMRealInstance( String n , APMAggregateDomainInstance a , APMPrimitiveDomain d , double val )
    {
        super( n , a , d );
        value = val;
        hasValue = true;

    }

    // Get Methods
    public double getRealValue()
    {
        int success = 0;
```

```
    // The instance has value; just return it.
    if( this.hasValue() )
        return value;

    // Inputs are supposed to have values, report an error
    if( this.isInput() )
        System.err.println( "ERROR: Input variable \"" + this.getAttributeName() + "\" has no value." );

    // This is an output without a value, attempt to solve for it
    success = this.trySolveForValue();

    if( success > 0 )
        return value;
    else
        System.err.println( "ERROR: Could not solve for value of " + this.getAttributeName() );

    // It will return a null
    return value;
}

public int trySolveForValue()
{
    ListOfConstraintNetworkRelations listOfConnectedRelations = new ListOfConstraintNetworkRelations();
    ConstraintNetworkRelation tempRelation;
    ListOfConstraintNetworkVariables tempListOfVariables = new ListOfConstraintNetworkVariables();
    ListOfStrings tempListOfVariableNames = new ListOfStrings();
    int instanceNumber;
    ListOfAPMPrimitiveDomainInstances listOfConnectedInstances = new ListOfAPMPrimitiveDomainInstances();
    boolean allVariablesAreInputs;
    APMPrimitiveDomainInstance tempInstance;
    APMRealInstance tempRealInstance;
    ListOfStrings listOfRelationsToSendToSolver = new ListOfStrings();
    ListOfStrings listOfVariablesWithValuesNames= new ListOfStrings();
    ListOfReals listOfVariablesValues = new ListOfReals();
    double resultValue;
    ListOfReals results;


    // Tries to solve for the value of this instance
    // If it finds a value, puts the value in "value" and sets "hasValue" to true
    // If the solver could not find a solution for the value, keeps "hasValue" as false

    // Returns the number of solutions found to be analyzed by the calling program accordingly


    // It makes no sense to try to solve for an input, just return true
    if( this.isInput() )
        return 1;

    // This instance is defined as an output. Solve for its value.
    System.out.println( "\nSolving for " + this.getFullAttributeName() );

    // Get the relations connected to this attribute
    // If there are no relations, just return 0 and display a message
    if( APMInterface.getConstraintNetwork().getNode( this.getFullAttributeName() ) == null )
    {
        System.out.println( "No solutions for " + this.getFullAttributeName() + " found." );
        return 0;
    }

    listOfConnectedRelations = APMInterface.getConstraintNetwork().getNode( this.getFullAttributeName()
).getConnectedRelations();

    // Eliminate those relations in which all variables have value
    for( int i = 0 ; i < listOfConnectedRelations.size() ; i++ )
    {
        // Clear list tempListOfVariableNames
```

```
tempListOfVariableNames = new ListOfStrings();

// Get a relation from listOfConnectedRelations
tempRelation = listOfConnectedRelations.elementAt( i );

// Get the variables DIRECTLY connected to tempRelation
tempListOfVariables = tempRelation.getVariables();

// Get the names of the variables in tempListOfVariables
for( int j = 0 ; j < tempListOfVariables.size() ; j++ )
    tempListOfVariableNames.addElement( tempListOfVariables.elementAt( j ).getName() );

// Get the instance number of this instance (useful in the case in which
// this instance is inside aggregates, and APMComplexDomainInstance.getInstances( name )
// returns several instances with the same same.
instanceNumber = this.getInstanceNumber( );

// Get the instances with the names contained in tempListOfVariableNames
listOfConnectedInstances = this.getConnectedInstances( tempListOfVariableNames , instanceNumber );

// Check if all these instances connected to this relation are inputs.
// If at least one instance IS NOT an input, set includeRelation to
// true (with the purpose of excluding those relations in which all
// variables are inputs)
allVariablesAreInputs = true;
for( int j = 0 ; j < listOfConnectedInstances.size() ; j++ )
{
    tempInstance = listOfConnectedInstances.elementAt( j );

    if( !tempInstance.isInput() )
    {
        allVariablesAreInputs = false;
        break;
    }

}

if( allVariablesAreInputs )
{
    // Display a warning message indicating that a relation has been ignored
    // because all its connected variables are inputs
    System.out.println( "WARNING: Ignoring relation \"" + tempRelation.getName() +
        "\" because all its variables are inputs. Values may not be consistent." );
    System.out.println( " Expression: " + tempRelation.getExpression() );
    for( int j = 0 ; j < tempListOfVariableNames.size() ; j++ )
        System.out.println( " " + tempListOfVariableNames.elementAt( j ) + " = " +
            ( (APMRealInstance) listOfConnectedInstances.elementAt( j ) ).getRealValue() );
}



// If flag includeRelation is true, include the relation and the variables in the lists
// that are going to be sent to the solver
if( !allVariablesAreInputs )
{
    // Add the expression of the relation
    listOfRelationsToSendToSolver.addElement( tempRelation.getExpression() );

    // Add the names of the variables with value and the values
    for( int j = 0 ; j < listOfConnectedInstances.size() ; j++ )
    {
        // Get and instance from listOfConnectedInstances
        tempInstance = listOfConnectedInstances.elementAt( j );

        // If, for any reason, tempInstance is not a real instance, send
        // and error message and exit returning 0.
        if( tempInstance.isAnAPMRealInstance() )
            tempRealInstance = (APMRealInstance) tempInstance;
```

```
            else
            {
                System.out.println( "Cannot solve, \"" + tempInstance.getFullAttributeName() + "\" has string value" );
                return 0;
            }


            if( tempRealInstance.isInput() )
            {
                // Add the name of the instance to listOfVariablesWithValuesNames (if is not already in the list)
                if( !listOfVariablesWithValuesNames.hasElement( tempRealInstance.getFullAttributeName() ) )
                {
                    listOfVariablesWithValuesNames.addElement( tempRealInstance.getFullAttributeName() );

                    // Add the value of the instance to listOfVariablesValues (if is not already in the list)
                    listOfVariablesValues.addElement( tempRealInstance.getRealValue() );
                }
            }
        }
    }
}

APMSolverWrapper solver = APMSolverWrapperFactory.makeSolverWrapperFor( "mathematica" );
APMSolverResult solverResults = solver.solveFor( this.getFullAttributeName() , listOfRelationsToSendToSolver ,
listOfVariablesWithValuesNames , listOfVariablesValues );


if( solverResults.hasResults() )
{
    // At least one solution was returned from the solver.

    // Get the list of results
    results = solverResults.getResults();
    System.out.println( "Solutions found: " + this.getFullAttributeName() + " = " + results.toString() );


    // Get the first positive value in the list of results (if any)
    // If there are no positive results, get the first result
    resultValue = results.elementAt( 0 );
    for( int k = 0 ; k < results.size() ; k++ )
        if( results.elementAt( k ) > 0 )
        {
            resultValue = results.elementAt( k );
            break;
        }

    // Display a warning message if multiple solutions were found.
    if( results.size() > 1 )
        System.out.println( "WARNING: Multiple solutions for " + this.getFullAttributeName() +
            " found, using first positive solution " + this.getFullAttributeName() + " = " + resultValue );


    // Set the value (which in turn sets hasValue to true)
    this.setValue( resultValue );

    return results.size();
}
else
{
    // Value not found
    System.out.println( "No solutions for " + this.getFullAttributeName() + " found." );
    return 0;
}


}
```

```
private ListOfAPMPrimitiveDomainInstances getConnectedInstances( ListOfStrings listOfConnectedVariableNames ,
int aggregateElementNumber )
{
    // Returns a list of instances connected to this one according to
    // the contraint network (INCLUDING this one)

    // aggregateElementNumber is given in case that "this" is inside an aggregate and
    // therefore there will be a LIST of connected instances for EACH element of the aggregate
    // The calling method must specify the element number in this case.

    // For example: there will be several instances of pwa.layup.thickness since layup
    // is an aggregate. Therefore, this method will return a list of all the
    // instances of pwa.layup.thickness (which will be APMRealInstances, in this case)

    ListOfAPMPrimitiveDomainInstances returnList = new ListOfAPMPrimitiveDomainInstances();
    ListOfAPMPrimitiveDomainInstances tempList;
    APMComplexDomainInstance rootInstance;
    String tempConnectedVariableName;
    ListOfStrings strippedTempConnectedVariableName;
    APMPrimitiveDomainInstance tempConnectedPrimitiveInstance = null;


    // Get the top instance that contains everybody (including this)
    rootInstance = this.getRootContainingInstance();


    for( int i = 0 ; i < listOfConnectedVariableNames.size() ; i++ )
    {
        // Get a name from the list (which is a full, dot-separated name)
        tempConnectedVariableName = listOfConnectedVariableNames.elementAt( i );

        // Strip the dot-separated name into a list of strings
        strippedTempConnectedVariableName = new ListOfStrings( tempConnectedVariableName );

        // Remove the first element of this list (the root domain name)
        strippedTempConnectedVariableName.removeElementAt( 0 );


        // Get all instances in rootInstance with the same strippedTempConnectedVariableName
        tempList = rootInstance.getInstances( strippedTempConnectedVariableName );


        // If we got more than one connected instance (because the connected instance
        // is inside an aggregate) get the aggregateElementNumber'th one in this list
        // that has the *same* fullAttributeName (there could be other instances in this
        // list corresponding to other domains that are subtypes of the same domain,
        // but we are not interested in those).
        // NOTE: this assumes that if we get more than one connected instance then
        // "this" and the connected instance are elements of the *SAME* aggregate.
        // (notice that aggregateElementNumber is the element number of "this", not
        // necesarily of the connected instance).



        tempConnectedPrimitiveInstance = tempList.elementAt( aggregateElementNumber );

        returnList.addElement( tempConnectedPrimitiveInstance );
    }

    return returnList;
}


private void resetConnectedOutputs()
{
    // Sets the hasValue flag to false of the instances connected to this one that
    // are outputs
```

```
    ListOfStrings listOfConnectedVariableNames;
    ListOfAPMPrimitiveDomainInstances listOfConnectedInstances;
    APMPrimitiveDomainInstance tempConnectedInstance;
    int instanceNumber;


    System.out.println( "Resetting values of connected output instances of \"" + this.getFullAttributeName() + "\"" );


    // Get the names of the instances connected to this one from the constraint network
    listOfConnectedVariableNames = APMInterface.getConstraintNetwork().getNode
    ( this.getFullAttributeName() ).getConnectedVariablesNames();

    // Get the instances connected to this one

    // Get the index of this instance in case it belongs to an aggregate
    instanceNumber = this.getInstanceNumber();

    listOfConnectedInstances = getConnectedInstances( listOfConnectedVariableNames , instanceNumber );


    // The following loop resets ALL connected attributes that are output. This may
    // be a little of an overkill, because the value of all connected attributes
    // that are outputs are not necessarily affected by this instance. But it
    // doesn't hurt, because they just will be calculated again.
    for( int i = 0 ; i < listOfConnectedInstances.size() ; i++ )
    {
        tempConnectedInstance = listOfConnectedInstances.elementAt( i );
        if( tempConnectedInstance.isOutput() )
        {
            System.out.println( "Resetting hasValue of " + tempConnectedInstance.getFullAttributeName() + " to False" );
            tempConnectedInstance.setHasValue( false );
        }
    }
}


public APMDomain getDomain()
{
    return (APMDomain) domain;
}


// Set Methods
public void setAsInput()
{
    System.out.println( "Setting \"" + this.getFullAttributeName() + "\" as input with value " + value );
    isInput = true;

}

public void setAsOutput()
{

    System.out.println( "Setting \"" + this.getFullAttributeName() + "\" as output" );

    // Reset the values of its former connected outputs
    this.resetConnectedOutputs();

    // Toggle isInput to false
    isInput = false;

    // Set hasValue as false. This is to be on the safe side in case
    // this instance is setAsOutput AFTER changing the value of some connected
    // input.
    // If this is setAsOutput BEFORE changing the value of a connected input, setting
    // hasValue as false here will be redundant, because APMRealInstance.setValue()
    // does it but it doesn't matter if we do it again here)
```

```
    System.out.println( "Resetting hasValue of \"" + this.getFullAttributeName() + "\" to false" );
    hasValue = false;

}


public void setValue( double v )
{
    // If "this" is an output
    if( this.isOutput() )
        System.out.println( "Setting value of output variable \"" + this.getFullAttributeName() + "\" = " + v );


    // If "this" is an input
    else if( this.isInput() )
    {
        // If "this" is an input with no value yet
        if( ! this.hasValue() )
            System.out.println( "Setting value of input variable \"" + this.getFullAttributeName() + "\" = " + v );

        // If "this" is an input with a previous value
        else if( this.hasValue() )
        {
            System.out.println( "Changing value of input variable \"" + this.getFullAttributeName() + "\" to " + v );

            // Must set the hasValue flag of the outputs connected to this input (if any) to false
            if( APMInterface.getConstraintNetwork().getNode( this.getFullAttributeName() ) != null )
                this.resetConnectedOutputs();
        }
    }

    value = v;
    hasValue = true;
}


public void setDomain( APMDomain dom )
{
    domain = (APMPrimitiveDomain) dom;
}


// Interrogation methods
public boolean isAnAPMRealInstance()
{
    return true;
}

public boolean hasValue()
{
    return hasValue;
}


// Information display methods
public String toString()
{
    DecimalFormat twoDecimalsFormat = new DecimalFormat();
    twoDecimalsFormat.setMaximumFractionDigits( 12 );

    // Input or output with value
    if( this.hasValue() )
        return String.valueOf( twoDecimalsFormat.format (this.getRealValue()) );

    // Input without a value
    else if( this.isInput() )
        return new String( "No value" );
```

```
        // Output whose value can be solved
        else if( this.trySolveForValue() != 0 )
            return String.valueOf( twoDecimalsFormat.format (this.getRealValue()) );

        // Output whose value cannot be solved
        return new String( "No value" );

    }

    public String toString( String indentString )
    {
        return this.toString();
    }


    // Create Copy Method
    public APMDomainInstance createCopy()
    {
        APMRealInstance returnInstance = new APMRealInstance( attributeName , domain );

        returnInstance.containedIn = this.containedIn;
        returnInstance.elementOf = this.elementOf;
        returnInstance.value = this.value;
        returnInstance.hasValue = this.hasValue;
        returnInstance.isInput = this.isInput;


        return returnInstance;
    }

}
```

APPENDIX H

APM PROTOCOL OPERATIONS PSEUDOCODES

M.1 Class APMInterface Operations[72]

APMInterface.loadAPMDefinitions

- Signature:

  APMInterface.loadAPMDefinitions(String **apmDefinitionFileName**)

- Local Variables:

  Boolean **success**

  APM **apmInstance**

- Procedure:

1 - **apmInstance** ← new APM(**apmDefinitionFileName**)

2 - setActiveAPM(**apmInstance**)

3 - **success** ← **apmInstance**.loadAPMDefinitions(**apmDefinitionFileName**)

4 - return **success**

Step 1 creates a new instance of **APM**. Step 2 sets this new instance as the active APM. Step 3 performs operation **APM.loadAPMDefinitions** (Appendix O) on this new instance. Step 4 returns the value of **success**.

N.1 Class APM Operations[73]

APM.loadAPMDefinitions

- Signature:

  APM.loadAPMDefinitions(String **apmDefinitionFileName**)

- Local Variables:

  APMParser **theAPMParser**

  APM **returnedAPM**

- Procedure:

1 - **theAPMParser** ← new APMParser(**apmDefinitionFileName**)

2 - **returnedAPM** ← **theAPMParser**.parse( )

---

[72] See the prototype implementation in Java of these operations in Appendix L.1.

[73] See the prototype implementation in Java of these operations in Appendix L.2.

3 ▸ **this.name** ← **returnedAPM**.getName( )

4 ▸ **this.sourceSets**← **returnedAPM**.getSourceSets( )

5 ▸ **this.sourceSetLinks**← **returnedAPM**.getSourceSetLinks( )

6 ▸ **this.linkedDomains**← **this.sourceSets**

7 ▸ linkAPMDefinitions( )

8 ▸ createConstraintNetwork( )

Step 1 creates a new instance of **APMParser**. Step 2 sends this instance a request to parse the APM Definition File given by **apm-DefinitionFile**. The instance of **APMParser** parses this file and returns an instance of class **APM,** which is stored in **returnedAPM**. Steps 3, 4, and 5 copy the contents of the **returnedAPM** (name, source sets and source set links, respectively) into the active APM (**this**). Step 6 copies each domain stored in **sourceSets** into **linkedDomains** (to be linked in Step 7). Step 7 links the domains from the different APM source sets using operation **APM.linkAPMDefinitions** (Appendix P). Step 8 creates the constraint network using operation **APM.createConstraintNetwork** (Appendix Q).

APM.linkAPMDefinitions

▸ Signature:

APM.linkAPMDefinitions( )

▸ Local Variables:

ListOfStrings **key1**, **key2**, **relativeKey1**, **relativeKey2**

String **sourceSetName1**, **sourceSetName2**, **domainName1**, **domainName2**, **keyLeaf1**, **keyLeaf2**

APMComplexDomain **domain1**, **domain2**, **insertionNode**, **insertedNode**

APMObjectDomain **objectDomainToBeUpdated**

APMAttribute **insertionAttribute**, **insertedAttribute**, **tempAttribute**

Integer **indexOfAttributeToBeUpdated**

ListOfAPMAttributes **tempAttributeList**, **tempLevels**

▸ Procedure:

1 ▸ for each APMSourceSetLink **tempSourceSetLink** ∈ **sourceSetLinks**:

2 ▸ ▸ **key1** ← **tempSourceSetLink**.getKeyAttribute1( ).getFullAttributeName( )

3 ▸ ▸ **key2** ← **tempSourceSetLink**.getKeyAttribute2( ).getFullAttributeName( )

4 ▸ ▸ **sourceSetName1** ← **key1**[0]

5 ▸ ▸ **sourceSetName2** ← **key2**[0]

6 ▸ ▸ **domainName1**← **key1**[1]

7 ▸ ▸ **domainName2**← **key2**[1]

8 ▸ ▸ **domain1**← **this.**getAPMDomain(**sourceSetName1** , **domainName1**)

```
9       ▸▸domain2← this.getAPMDomain( sourceSetName2 , domainName2 )
10      ▸▸relativeKey1← key1 minus source set name and domain set name
11      ▸▸relativeKey2← key2  minus source set name and domain set name
12      ▸▸keyLeaf1← key1[ last ]
13      ▸▸keyLeaf2← key2[ last ]
14      ▸▸insertionAttribute ← domain1.getAttribute( relativeKey1 )
15      ▸▸insertedAttribute ← domain2.getAttribute( relativeKey2 )
16      ▸▸insertionNode ← insertionAttribute.getContainerDomain( )
17      ▸▸insertedNode ← insertedAttribute.getContainerDomain( )
18      ▸▸if  insertionNode is an APM Object Domain
19        ▸▸▸objectDomainToBeUpdated ← insertionNode.getSupertypeDomainContainingAttribute(
          keyLeaf1 )
20        ▸▸▸indexOfAttributeToBeUpdated ← objectDomainToBeUpdated.getIndexOfLocalAttribute(
          keyLeaf1 )
21        ▸▸▸if insertedNode is an APM Object Domain
22          ▸▸▸▸tempAttributeList ← objectDomainToBeUpdated.getLocalAttributes( )
23          ▸▸▸▸tempAttribute ← new APMObjectAttribute( keyLeaf1 , objectDomainToBeUpdated ,
            insertedNode )
24          ▸▸▸▸ tempAttributeList[ indexOfAttributeToBeUpdated ] ← tempAttribute
25        ▸▸▸else  if insertedNode is an APM Multi-Level Domain
26          ▸▸▸▸tempAttributeList ← objectDomainToBeUpdated.getLocalAttributes( )
27          ▸▸▸▸tempAttribute ← new APMMultiLevelAttribute( keyLeaf1 , objectDomainToBeUpdated ,
            insertedNode )
28          ▸▸▸▸ tempAttributeList[ indexOfAttributeToBeUpdated ] ← tempAttribute
29      ▸▸else  if insertionNode is an APM Multi-Level Domain
30        ▸▸▸indexOfAttributeToBeUpdated ← insertionNode.getIndexOfLevel( keyLeaf1 )
31        ▸▸▸if insertedNode is an APM Object Domain
32          ▸▸▸▸tempLevels ← insertionNode.getLevels( )
33          ▸▸▸▸tempAttribute ← new APMObjectAttribute( keyLeaf1 , insertionNode , insertedNode
            )
34          ▸▸▸▸tempLevels[ indexOfAttributeToBeUpdated ] ← tempAttribute
35        ▸▸▸else if  insertedNode is an APM Multi-Level Domain
36          ▸▸▸▸tempLevels ← insertionNode.getLevels( )
37          ▸▸▸▸tempAttribute ← new APMMultiLevelAttribute( keyLeaf1 , insertionNode ,
            insertedNode )
38          ▸▸▸▸tempLevels[ indexOfAttributeToBeUpdated ] ← tempAttribute
```

Step 1 gets an **APMSourceSetLink** from list **sourceSetLinks** of the **APM**. Steps 2 and 3 get the full names of the key attributes of the source set link. Steps 4 and 5 get the names of their corresponding source sets, stored in the first position of **key1** and **key2**. Steps 6 and 7 get the names of the root domains that indirectly contain the key attributes, stored in the second position of **key1** and **key2**. Steps 8 and 9 get both root domains (the top domains that contain the attributes being linked) from their respective source sets (operation **getAPMDomain** gets the domains from list **linkedDomains**). Steps 10 and 11 build two lists of strings by removing the source set names and the domain set names from **key1** and **key2**. Steps 12 and 13 get the last elements of lists **key1** and **key2** (corresponding to the attribute names of the link attributes). Step 14 gets the **APMAttribute** with name **relativeKey1** from **domain1** (the "insertion attribute"). Step 15 gets the **APMAttribute** with name **relativeKey2** from **domain2** (the "inserted attribute"). Step 16 gets the **APMComplexDomain** that directly contains **insertionAttribute** (the "insertion node"). Step 17 gets the **APMComplexDomain** that directly contains **insertedAttribute** (the "inserted node"). If the **insertionNode** is an **APMObjectDomain**, Step 19 gets the domain that contains the **insertionAttribute** (since **insertionNode** is an **APMObjectDomain**, the domain that contains **insertionAttribute** could be either a supertype of **insertionNode** – if the **insertionAttribute** is inherited – or **insertionNode** itself if it is local). Step 20 gets the index of the insertion attribute. Steps 21 – 24 replace the insertion attribute with the inserted attribute when the **insertedNode** is an **APMObjectDomain** and Steps 25 – 28 when the **insertedNode** is an **APMMultiLevelDomain**. Steps 29 – 38 do the same as Steps 19 – 28 for the case when **insertionNode** is an **APMMultiLevelDomain**.

Figures M-1 through M-4 show how the variables used in the pseudocode of operation **APM.linkAPMDefinitions** correspond to the different APM objects for the source set links of the example of Subsection 59. The resulting linked APM is shown in Figure M-5.

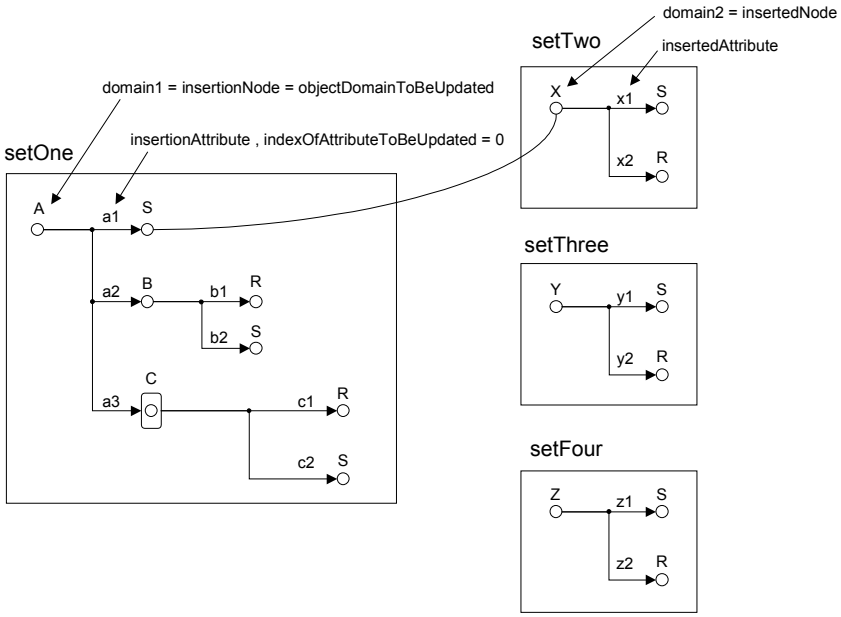


Figure M-1: Source Set Link Example: Source Set Link 1

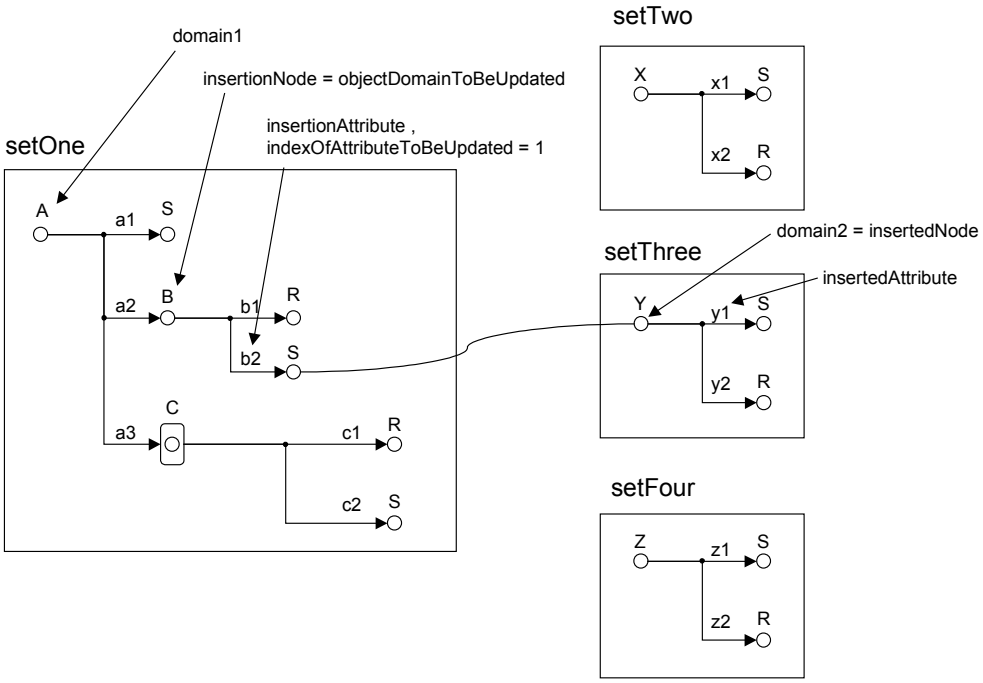


Figure M-2: Source Set Link Example: Source Set Link 2

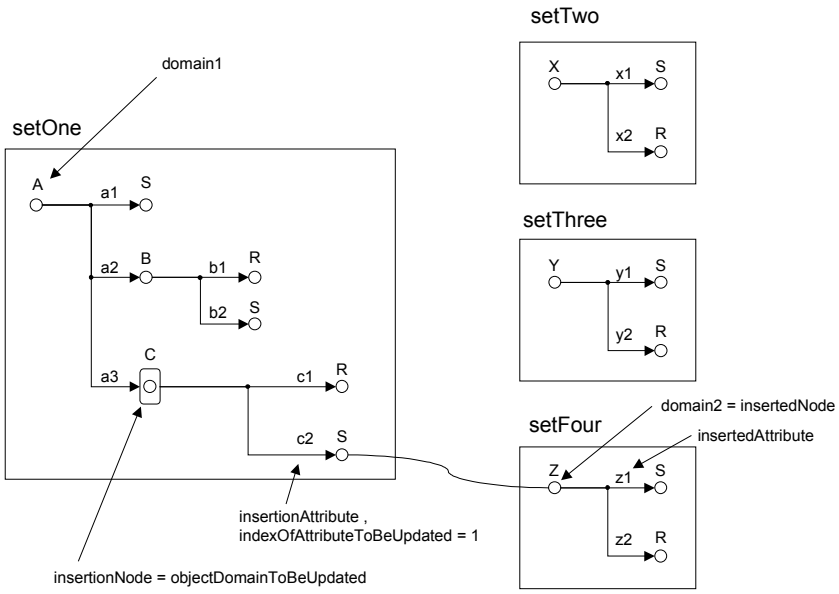


Figure M-3: Source Set Link Example: Source Set Link 3

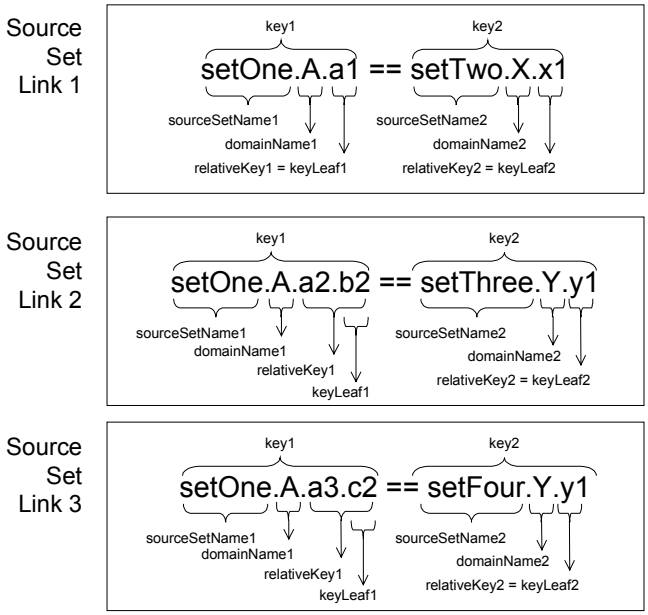


Figure M-4: Source Set Link Example: Source Set Link Definitions

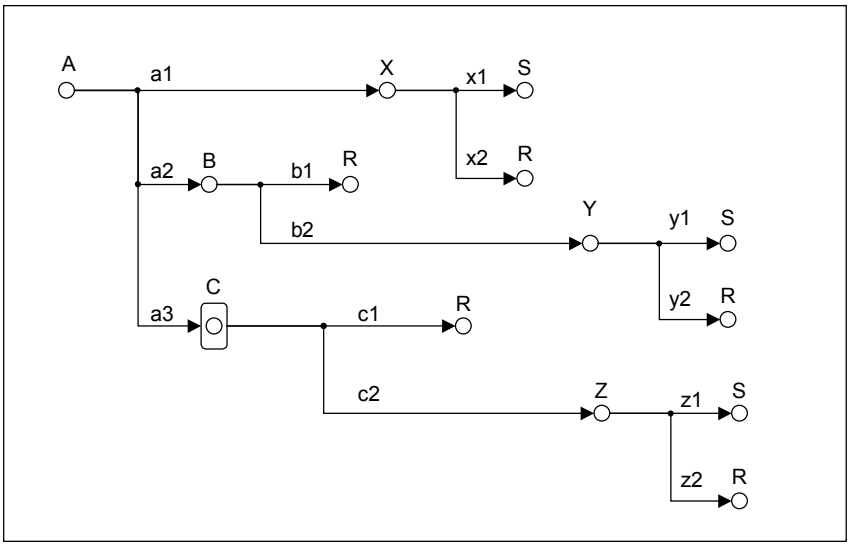


Figure M-5: Source Set Link Example: Resulting Linked APM

APM.createConstraintNetwork

- Signature:

  APM.createConstraintNetwork( )

- Local Variables:

  APMDomain **tempAPMDomain**

  String **domainName**

- Procedure:

1 - **this**.**constraintNetwork** ← new ConstraintNetwork( )
2 - for each APMDomain **tempAPMDomain** ∈ **linkedDomains**
3 - - if **tempAPMDomain** is an APMComplexDomain
4 - - - **domainName** ← **tempAPMDomain**.getDomainName( )
5 - - - addRelationsToConstraintNetwork(**domainName** , **tempAPMDomain**)

Step 1 creates a new instance of **ConstraintNetwork** and assigns it to variable **constraintNetwork** of the active APM (**this**). Step 2 gets an **APMDomain** from the list of **linkedDomains**. If **tempAPMDomain** is an **APMComplexDomain**, then Step 4 gets its name and Step 5 adds its relations to the constraint network with operation **APM.addRelationsToConstraintNetwork** (Appendix R), where the actual creation of the constraint network occurs.

APM.addRelationsToConstraintNetwork

- Signature:

  APM.addRelationsConstraintNetwork(String **nameSuffix** , APMComplexDomain **d**)

- Local Variables:

  ListOfAPMRelations **localRelations**

  String **relationName**, **relation**, **tempAttributeName**

  ConstraintNetworkRelation **tempCNRelation**

  ListOfAPMAttributes **listOfAttributes**

  APMDomain **tempAttributeDomain**

- Procedure:

```
1   localRelations ← d.getLocalRelations( )
2   for each APMRelation tempRelation ∈ localRelations:
3       relationName ← tempRelation.getRelationName( )
4       relation ← tempRelation.getRelation( )
5       if tempRelation is an APM Product Relation
6           tempCNRelation ← new ConstraintNetworkRelation( nameSuffix , relationName ,
            this.constraintNetwork , relation , "Product Relation" )
7       else if tempRelation is an APM Product Idealization Relation
8           tempCNRelation ← new ConstraintNetworkRelation( nameSuffix , relationName ,
            this.constraintNetwork , relation , "Product Idealization Relation" )
9       this.constraintNetwork.addRelation( tempCNRelation )
10      if d is an APM Object Domain:
11          listOfAttributes ← d.getAttributes( )
12      else if d is an APM Multi-Level Domain:
13          listOfAttributes ← d.getLevels( )
14      for each APMAttribute tempAttribute ∈ listOfAttributes:
15          tempAttributeName ← tempAttribute.getAttributeName( )
16          tempAttributeDomain ← tempAttribute.getAttributeDomain( )
17          if tempAttribute.getDomain is an APM Complex Domain:
18              addRelationsToConstraintNetwork( nameSuffix + "." + tempAttributeName ,
                tempAttributeDomain )
19          else if tempAttribute.getDomain is an APM Complex Aggregate Domain:
20              addRelationsToConstraintNetwork( nameSuffix + "." + tempAttributeName ,
                tempAttributeDomain.getDomainOfElements( ) )
```

Step 1 gets the local relations from the **APMComplexDomain d**. Step 2 gets an **APMRelation** from **localRelations**. Steps 3 and 4 get the name and the expression of **tempRelation**. If **tempRelation** is an **APMProductRelation**, then Step 6 creates a new **ConstraintNetworkRelation** of type "Product Relation". Else if **tempRelation** is an **APMProductIdealizationRelation**, then Step 8 creates a new **ConstraintNetworkRelation** of type "Product Idealization Relation". It is in Steps 6 and 8 where the actual creation of the constraint network occurs, when the new ConstraintNetworkRelation is created with the **ConstraintNetworkRelation** constructor explained in Appendix X. Step 9 adds **tempCNRelation** to the constraint network. If **d** is an **APMObjectDomain**, then Step 11 gets its attributes. Else if **d** is an **APMMultiLevelDomain**, Step 13 gets its levels. Steps 14 – 20 recursively perform this operation on each complex attribute in **listOfAttributes**, adding a dot followed by the attribute name to **nameSuffix** in **addRelationsToConstraint-Network**.

APM.loadSourceSetData

- Signature:

  APM.loadSourceSetData(ListOfStrings **listOfFileNames**)

- Local Variables:

  ListOfAPMSourceSets **listOfSourceSets**

  Integer **i** ← 0

  String **tempFileName**, **tempSourceSetName**

  APMSourceDataWrapperObject **wrapperObject**

  APMComplexDomain **tempSourceSetRootDomain**

  ListOfAPMComplexDomains **listOfSourceSetRootDomainSubtypes**

  ListOfAPMSourceDataWrapperReturnedObjects **returnedListOfObjects**

  APMComplexDomainInstance **instance**, **instanceCopy**

  ListOfAPMDomainInstances **listOfSourceSetInstances**, **listOfLinkedSourceSetInstances**

- Procedure:

1 **listOfSourceSets** ← **this**.getSourceSets( )

2 for each APMSourceSet **tempSourceSet** ∈ **listOfSourceSets**:

3 **tempSourceSetName** ← **tempSourceSet**.getSourceSetName( )

4 **tempFileName** ← **listOfFilesNames**[**i**]

5 **i** ← **i** + 1

6 **wrapperObject** ← APMSourceDataWrapperFactory.makeWrapperObjectFor(**tempSourceSetName** , **tempFileName**)

7 **tempSourceSetRootDomain** ← **tempSourceSet**.getRootDomain( )

```
8       ▸▸listOfSourceSetRootDomainSubtypes ← tempSourceSet.getSubtypesOf(
        tempSourceSetRootDomain )
9       ▸▸for each APMComplexDomain tempSourceSetRootDomainSubtype ∈
        listOfSourceSetRootDomainSubtypes:
10          ▸▸▸returnedListOfObjects ← wrapperObject.getInstancesOf(
            tempSourceSetRootDomainSubtype )
11          ▸▸▸for each APMSourceDataWrapperReturnedObject tempReturnedObject ∈
            returnedListOfObjects:
12              ▸▸▸▸if tempSourceSetRootDomain is an APM Object Domain:
13                  ▸▸▸▸▸instance ← new APMObjectDomainInstance( "root" ,
                    tempSourceSetRootDomainSubtype )
14                  ▸▸▸▸▸instanceCopy ← new APMObjectDomainInstance( "root" ,
                    tempSourceSetRootDomainSubtype )
15              ▸▸▸▸else if tempSourceSetRootDomain is an APM Multi-Level Domain:
16                  ▸▸▸▸▸instance ← new APMMultiLevelDomainInstance( "root" ,
                    tempSourceSetRootDomainSubtype )
17                  ▸▸▸▸▸instanceCopy ← new APMMultiLevelDomainInstance( "root" ,
                    tempSourceSetRootDomainSubtype )
18          ▸▸▸instance.populateWithValues( tempReturnedObjects.getValues( ) )
19          ▸▸▸instanceCopy.populateWithValues( tempReturnedObjects.getValues( ) )
20          ▸▸▸listOfSourceSetInstances ← tempSourceSet.getSetInstances( )
21          ▸▸▸listOfSourceSetInstances.addElement( instance )
22          ▸▸▸listOfLinkedSourceSetInstances ← tempSourceSet.getLinkedInstances( )
23          ▸▸▸listOfsourceSetInstances.addElement( instanceCopy )
24      ▸this.linkSourceSetData( )
```

Step 1 gets the list of **APMSourceSets** stored in the active APM (**this**). Step 2 gets an **APMSourceSet** from this list. Step 3 gets the name of **tempSourceSet**. Step 4 gets the name of the repository where the data for this source set is stored. Step 6 asks the **APMSourceDataWrapperFactory** to create an instance of **APMSourceDataWrapperObject** (see Appendix W). Step 7 gets the root domain of this source set. Step 8 gets a list of all domains in this subset that are subtyped (directly or indirectly) from **tempSourceSetRootDomain** (the list will include the root domain itself). Step 9 gets an **APMComplexDomain** from **listOf-SourceSetRootDomainSubtypes**. Step 10 asks **wrapperObject** to get the instances of **tempSourceSetRootDomain-Subtype** stored in the repository. Step 11 gets an **APMSourceDataWrapperReturnedObject** from **returnedListOf-Objects**. If **tempSourceSetRootDomainSubtype** is an **APMObjectDomain**, then Steps 13 and 14 create two instances of **APMObjectDomainInstance**. Else if **tempSourceSetRootDomainSubtype** is an **APMMultiLevelDomain**, then Steps 16 and 17 create two instances of **APMMultiLevelDomainInstance**. Steps 18 and 19 fill **instance** and **instanceCopy** with the values returned by the wrapper (contained in **tempReturnedObject**). Steps 20 and 21 add **instance** to **listOfSource-**

SetInstances of the set. Steps 22 and 23 add **instanceCopy** to **listOfSourceSetLinkedInstances** of the **APM** (to be linked in Step 24). Step 24 performs operation **APM.linkSourceSetData** (Appendix T) to link the data.

APM.linkSourceSetData

- Signature:

  APM.linkSourceSetData( )

- Local Variables:

  ListOfStrings **fullKeyAttributeName1**, **fullKeyAttributeName2**, **keyAttributeName1**, **keyAttributeName2**

  String **sourceSetName1**, **sourceSetName2**, **domainName1**, **domainName2**, **lastKeyAttributeName1**, **lastKeyAttributeName2**

  APMSourceSet **sourceSet1**, **sourceSet2**

  APMComplexDomain **domain1**, **domain2**

  ListOfAPMComplexDomainInstances **listOfDomain1Instances**, **listOfDomain2Instances**, **containingInstance1Values, containingInstance2Values**

  ListOfAPMPrimitiveDomainInstances **listOfKey1Instances**, **listOfKey2Instances**

  APMComplexDomainInstance **containingInstance1**, **containingInstance2**, **key1InstanceCopy**, **containingInstance2Copy**, **emptyDomain2Instance**

  Integer **attributeIndex1**, **attributeIndex2**

  Boolean **matchFound**

- Procedure:

1 - for each APMSourceSetLink **tempSourceSetLink** ∈ **sourceSetLinks**:

2 - - **fullKeyAttributeName1** ← **tempSourceSetLink**.getKeyAttribute1( ).getFullAttributeName( )

3 - - **fullKeyAttributeName2** ← **tempSourceSetLink**.getKeyAttribute2( ).getFullAttributeName( )

4 - - **sourceSetName1** ← **fullKeyAttributeName1**[0]

5 - - **sourceSetName2** ← **fullKeyAttributeName2**[0]

6 - - **domainName1** ← **fullKeyAttributeName1**[1]

7 - - **domainName2** ← **fullKeyAttributeName2**[1]

8 - - **lastKeyAttributeName1** ← **fullKeyAttributeName1**[last]

9 - - **lastKeyAttributeName2** ← **fullKeyAttributeName2**[last]

10 - - **keyAttributeName1** ← **fullKeyAttributeName1** – first and second elements

11 - - **keyAttributeName2** ← **fullKeyAttributeName2** – first and second elements

12 - - **sourceSet1** ← **this**.getSourceSet(**sourceSetName1**)

13 - - **sourceSet2** ← **this**.getSourceSet(**sourceSetName2**)

14 - - **domain1** ← **sourceSet1**.getDomain(**domainName1**)

15 ▸▸ **domain2** ← **sourceSet2**.getDomain(**domainName2**)

16 ▸▸ **listOfDomain1Instances** ← **this**.getInstancesOf(**sourceSetName1** , **domainName1**)

17 ▸▸ **listOfDomain2Instances** ← **this**.getInstancesOf(**sourceSetName2** , **domainName2**)

18 ▸▸ for each APMComplexDomainInstance **tempDomain1Instance** ∈ **listOfDomain1Instances**:

19 ▸▸▸ **listOfKey1Instances** ← **tempDomain1Instance**.getInstances(**keyAttributeName1**)

20 ▸▸▸ for each APMPrimitiveDomainInstance **key1Instance** ∈ **listOfKey1Instances**:

21 ▸▸▸▸ **matchFound** ← **false**

22 ▸▸▸▸ for each APMComplexDomainInstance **tempDomain2Instance** ∈ **listOfDomain2Instances**:

23 ▸▸▸▸▸ if **matchFound** = **true** go to 20

24 ▸▸▸▸▸ **listOfKey2Instances** ← **tempDomain2Instance**.getInstances(**keyAttributeName2**)

25 ▸▸▸▸▸ for each APMPrimitiveDomainInstance **key2Instance** ∈ **listKey2Instances**:

26 ▸▸▸▸▸▸ if (**key1Instance** is an APM String Instance AND **key1Instance**.getStringValue( ) = **key2Instance**.getStringValue( ))

OR

(**key1Instance** is an APM Real Instance AND **key1Instance**.getRealValue( ) = **key2Instance**.getRealValue( ))

27 ▸▸▸▸▸▸▸ **containingInstance1** ← **key1Instance**.getContainedIn( )

28 ▸▸▸▸▸▸▸ **key1InstanceCopy** ← **key1Instance**.createCopy( )

29 ▸▸▸▸▸▸▸ **containingInstance1**.addCopyOfKeyValueBeforeLinking(**key1InstanceCopy**)

30 ▸▸▸▸▸▸▸ **attributeIndex1** ← **containingInstance1**.getIndexOf(**lastKeyAttributeName1**)

31 ▸▸▸▸▸▸▸ **containingInstance2** ← **key2Instance**.getContainedIn( )

32 ▸▸▸▸▸▸▸ **containingInstance2Copy** ← **containingInstace2**.createCopy( )

33 ▸▸▸▸▸▸▸ **containingInstance2Copy**.setAttributeName(**lastKeyAttributeName1**)

34 ▸▸▸▸▸▸▸ **containingInstance1Values** ← **containingInstance1**.getValues( )

35 ▸▸▸▸▸▸▸ **containingInstance1Values**[**attributeIndex1**] ← **containingInstance2Copy**

36 ▸▸▸▸▸▸▸ **containingInstance2Copy**.setContainedIn(**containingInstance1**)

37 ▸▸▸▸▸▸▸ **matchFound** ← **true**

38 ▸▸▸▸▸▸▸ go to 22

39 ▸▸▸▸ **containingInstance1** ← **key1Instance**.getContainedIn( )

40 ▸▸▸▸ **attributeIndex1**← **containingInstance1**.getIndexOf(**lastKeyAttributeName1**)

41 ▸▸▸▸ if **domain2** is an APM Object Domain

42 ▸▸▸▸▸ **emptyDomain2Instance** ← new APMObjectDomainInstance(**lastKeyAttributeName1** , **containingInstance1** , **domain2**)

43 ▸▸▸▸ else if **domain2** is an APM Multi-Level Domain

44 ▸▸▸▸▸ **emptyDomain2Instance** ← new APMMultiLevelDomainInstance(**lastKeyAttributeName1** , **containingInstance1** , **domain2**)

45 ▸▸▸▸▸ **emptyDomain2Instance**.instantiateAllAttributes( )

46 ▸▸▸▸▸ **key1Instance**.setAttributeName(**lastKeyAttributeName2**)

47 ▸▸▸▸▸ **containingInstance1Values** ← **containingInstance1**.getValues( )

48 ▸▸▸▸▸ **containingInstance1Values**[**attributeIndex1**] ← **emptyDomain2Instance**

Step 1 gets an **APMSourceSetLink** from list **sourceSetLinks** of the active APM (**this**). Steps 2 and 3 get the full attribute names of the two key attributes of the source set link. Steps 4 and 5 get the source set names. Steps 6 and 7 get the names of the root domains that indirectly contain the key attributes. Steps 8 and 9 get the names of the key attributes. Steps 10 and 11 create two new list of strings by removing the first and second elements (corresponding to the source set name and the domain name) from lists **fullKeyAttributeName1** and **fullKeyAttributeName2**. Steps 12 and 13 get the two source sets. Steps 14 and 15 get the two root domains. Step 16 gets all instances of **domain1** in **sourceSet1**. Step 17 gets all instances of **domain2** in **sourceSet2**. Step 18 gets an **APMComplexDomainInstance** from **listOfDomain1Instances**. Step 19 gets the instances of the attribute called **keyAttributeName1** contained by **tempDomain1Instance** (there could be more than one if the attribute is inside an aggregate). Step 20 gets an **APMPrimitiveDomainInstance** from **listOfKey1Instances**. Step 21 resets the flag **matchFound** to **false**. Step 22 gets an **APMPrimitiveDomainInstance** from **listOfKey2Instances**. In Step 23, if a match has been found, break the loop and go back to Step 20. Else, Step 24 gets the instances of the attribute called **keyAttributeName2** contained by **tempDomain2Instance** (there could be more than one if the attribute is inside an aggregate). Step 25 gets an **APMPrimitiveDomainInstance** from **listKey2Instances**. Step 26 compares **key1Instance** and **key2Instance**. If the two instances match, Step 27 gets the instance that contains **key1Instance**. Step 28 creates a copy of **key1Instance**. Step 29 stores a copy of **key1Instance** in variable **copiesOfKeyValuesBeforeLinking** of **containingInstance1** (this is used for the unlink operation explained in Subsection 0). Step 30 gets the position of **key1Instance** in **containingInstance1**. Step 31 gets the instance that directly contains **key2Instance**. Step 32 creates a copy of **containingInstance2**. Step 33 changes the attribute name of **containingInstance2Copy** to the attribute name of the insertion attribute (**lastKeyAttributeName1**). Step 34 gets the values contained in **containingInstance1**. Step 35 points the value in position **attributeIndex1** of **containingInstance1Values** to **containingInstance2Copy**[74]. Step 36 sets variable **containedIn** of

[74] **containingInstance1** is linked to a *copy* of **containingInstance2** to allow multiple instance from the first source set to be linked with the same instance of the second source set. If the original instance is used, then the attribute name set in the previous step will be inconsistent.

containingInstance2Copy to point to **containingInstance1** (which now contains **containingInstance2Copy**). Step 37 sets flag **matchFound** to **true**. Step 38 returns to Step 22 for another iteration. If the program executes past Step 38, it means that a match for **key1Instance** was not found in **listOfDomain2Instances**, and therefore **key1Instance** will be replaced with an empty instance of **domain2**. For this purpose, Step 39 gets the instance that contains **key1Instance**. Step 40 gets the position of **key1Instance** in **containingInstance1**. Steps 42 and 44 create a new instance of **domain2**, depending on whether **domain2** is an **APMObjectDomain** or an **APMMultiLevelDomain**, respectively. Step 45 fills this new instance with empty values. Step 46 changes the attribute name of **key1Instance** to **lastKeyAttributeName2**. Steps 47 and 48 point the attribute in position **attributeIndex1** of **containingInstance1** to the new **emptyDomain2Instance**.

As an aid for the pseudocode above, Figure M-6 shows how two instances from the example of Subsection 60 are linked - according to the third source set link defined in the example – indicating some of the variables involved in this operation.

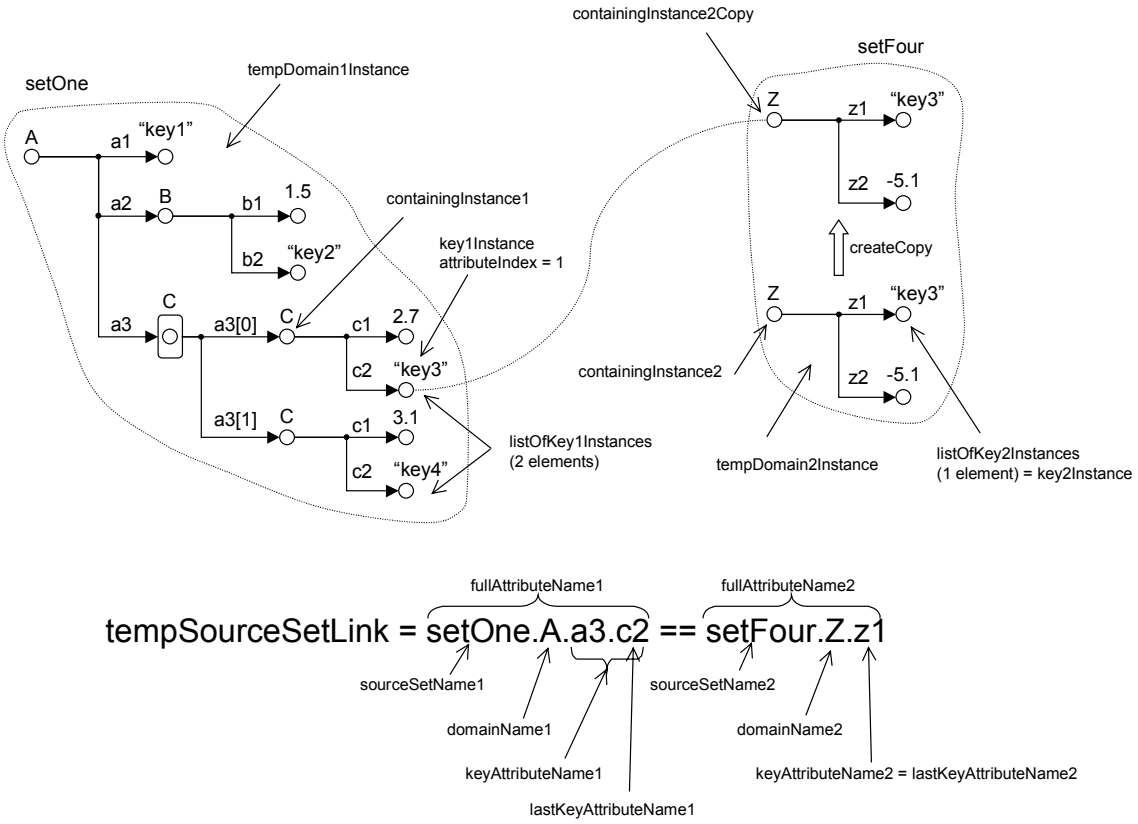


Figure M-6: Operation **APM.linkSourceSetData** Example Showing Variables Involved

T.1 APMRealInstance Class Operations[75]

APMRealInstance.getRealValue

- Signature:

  APMRealInstance.getRealValue( )

- Local Variables:

  Boolean **success**

- Procedure:

```
1   if this.hasValue( )
2       return this.value
3   if this.isInput( )
4       Print an error message indicating that this input instance has no value
5       return NIL
6   success ← this.trySolveForValue( )
7   if success > 0
8       return success
9   else
10      Print message indicating that the value could not be found
11      return NIL
```

If this **APMRealInstance** has value, Step 2 simply returns it. Else, if it does not have a value and is an input, then Step 4 prints an error message and Step 5 returns **NIL** (if an **APMRealInstance** is declared as an input, it should have value). Else, if the instance does not have value and is an output, then Step 6 will use operation **APMRealInstance.trySolveForValue** (Appendix V) to try to solve for its value. If the operation is successful, the value of **success** will be positive, and therefore Step 8 will returns it. Otherwise, Step 10 will print a message indicating that the value could not be found and Step 11 will return **NIL**.

APMRealInstance.trySolveForValue

- Signature:

  APMRealInstance.trySolveForValue( )

- Local Variables:

  ConstraintNetwork **constraintNetwork**

  ConstraintNetworkNode **constraintNetworkVariableNode**

[75] See the prototype implementation in Java of these operations in Appendix L.3.

ListOfConstraintNetworkRelations **listOfConnectedRelations**, **listOfRelationsToSendToSolver**

ListOfStrings **tempListOfVariableNames**, **listOfVariablesWithValueNames**

ListOfConstraintNetworkVariables **tempListOfVariables**

Integer **instanceNumber**

ListOfAPMPrimitiveDomainInstances **listOfConnectedInstances**

Boolean **allVariablesAreInputs**

APMSolverWrapper **solver**

APMSolverResult **solverResult**

Real **resultValue**

ListOfReals **results**, **listOfVariablesValues**

▸ Procedure:

1 ▸ if **this**.isInput( )

2 ▸ ▸ return **1**

3 ▸ **constraintNetwork** ← APMInterface.getConstraintNetwork( )

4 ▸ **constraintNetworkVariableNode** ← **constraintNetwork**.getNode(**this**.fullAttributeName( ))

5 ▸ **listOfConnectedRelations** ← **constraintNetworkVariableNode**.getConnectedRelations( )

6 ▸ **listOfRelationsToSendToSolver** ← relations in **listOfConnectedRelations** in which at least one variable is an output

7 ▸ for each ConstraintNetworkRelation **tempRelation** ∈ **listOfConnectedRelations**:

8 ▸ ▸ **tempListOfVariables** ← **tempRelation**.getVariables( )

9 ▸ ▸ for each String **tempVariable** ∈ **tempListOfVariables**:

10 ▸ ▸ ▸ **tempListOfVariableNames**[last] ← **tempVariable**.getName( )

11 ▸ ▸ **instanceNumber** ← **this**.getInstanceNumber( )

12 ▸ ▸ **listOfConnectedInstances** ← getConnectedInstances(**tempListOfVariableNames** , **instanceNumber**)

13 ▸ ▸ for each APMPrimitiveDomainInstance **tempInstance** ∈ **listOfConnectedInstances**:

14 ▸ ▸ if **tempInstance**.isAnOutput( ) = **true**

15 ▸ ▸ ▸ **allVariablesAreInput** ← **false**

16 ▸ ▸ if **allVariablesAreInput** = **true**

17 ▸ ▸ ▸ Print a warning indicating that **tempRelation** is being ignored from the system of equations being sent to the solver

18 ▸ ▸ else if **allVariablesAreInput** = **false**

19 ▸ ▸ ▸ **listOfRelationsToSendToSolver**[last] ← **tempRelation**.getExpression( )

20 ▸ ▸ ▸ Add the *name* of each input variable in **listOfConnectedInstances** to **listOfVariablesWithValueNames**

```
21          ▸ ▸ ▸ Add the value of each input variable in listOfConnectedInstances to listOfVariablesValues
22    ▸ solver ← APMSolverWrapperFactory.makeSolverWrapperFor( "mathematica" )
23    ▸ solverResult ← solver.solveFor( this.getFullAttributeName( ) , listOfRelationsToSendToSolver ,
      listOfVariablesWithValuesNames , listOfVariablesValues )
24    ▸ if solverResults.hasResults( ) = true
25       ▸ ▸ results ← solverResults.getResults( )
26       ▸ ▸ if there is at least one positive element in list results
27          ▸ ▸ ▸ resultValue ← first positive element in list results
28       ▸ ▸ else if there are no positive elements in list results
29          ▸ ▸ ▸ resultValue ← first element in list results
30       ▸ ▸ this.setValue( resultValue )
31       ▸ ▸ return results.size( )
32    ▸ else if solverResults.hasResults( ) = false
33       ▸ ▸ Print a message indicating that no solutions for this attribute were found
34       ▸ ▸ return 0
```

If the **APMRealInstance** is an input, then Step 2 just returns **1** and exits (it makes no sense to try to solve for the value of an input). Else, Step 3 gets the constraint network of the active APM. Step 4 gets the **ConstraintNetworkVariable** in the constraint network corresponding to this instance. Step 5 gets the **ConstraintNetworkRelations** connected to this **ConstraintNetworkVariable**. Step 6 builds a list of **ConstraintNetworkRelations** with the **ConstraintNetworkRelations** in **listOfConnectedRelations** in which at least *one* of the participating variables is an input. Relations in which *all* variables are inputs are ignored since they do not add any new information to the system of equations and also because they could potentially introduce inconsistencies caused by round-off errors. For example, consider the relation $\mathbf{a} = \pi \mathbf{r}^2$. Since π is an irrational number, it will be practically impossible to provide input values for **r** and **a** such that **a** is *exactly* $\pi \mathbf{r}^2$. Therefore, a constraint solver may interpret this apparent inconsistency as if the system of equations has no solutions[76]. Step 7 gets a **ConstraintNetworkRelation** from **listOfConnectedRelations**. Step 8 gets the **ConstraintNetworkVariables** *directly* connected to **tempRelation**. Steps 9 and 10 copy the names of each **ConstraintNetworkVariable** in **tempListOfVariables** into a new list. Step 11 gets the index of this instance, which may be different from **0** only when this **APMRealInstance** is inside an aggregate instance. In this case, this index will allow differentiating this **APMRealInstance** from the others in the same aggregate. Step 12 gets the instances that – according to the constraint network – are connected to this **APMRealInstance**. Steps 13, 14 and 15 check whether all the **APMPrimitiveDomainInstances** in **listOfConnectedInstances** are inputs. If not, the flag **allVariablesAreInput** is set to **false**. If all the variables participating in the relation are inputs, Step 17 prints a warning message indicating that **tempRelation** is being ignored from the system of equations being sent to the solver. Else, if at least one variable is an output, Step 19

[76] This problem could also be solved by specifying some kind of tolerance to determine equality between two values.

adds the expression of **tempRelation** to the **listOfRelationsToSendToSolver**. Step 20 adds the *name* of each input variable in **listOfConnectedInstances** to list **listOfVariablesWithValueNames**, and Step 21 adds their *value* to list **listOfVariableValues**. Step 22 creates an **APMSolverWrapper** for the external constraint solver being used (Mathematica in this case). Step 23 sends the name of the variable being solved, the list of relations, the list of variable names and the list of variable values to the constraint solver (via the **APMSolverWrapper solver**). Step 24 checks if the value returned by **solver** has results. If so, Step 25 extracts them and puts them in a list of real numbers. If there is at least one positive result in **results**, then Step 27 assigns the value of the first positive result to **resultValue**. Else, if there are no positive values in **results**, then Step 29 assigns the first result to **resultValue**. Step 30 sets the value of this **APMRealInstance** to **resultValue**. Step 31 returns the number of results found. If no results where found, then Step 33 prints a message indicating that no solutions for this attribute were found and Step 34 returns **0**.

V.1 APMSourceDataWrapperFactory Class Operations

APMSourceDataWrapperFactory.makeWrapperObjectFor

- Signature:

  APMSourceDataWrapperFactory.makeWrapperObjectFor(String **sourceSetName** , String **dataFileName**)

- Local Variables:

  String **formatName**

- Procedure:

1 - **formatName** ← this.getDataFormatForSourceSet(sourceSetName)
2 - if **formatName** ← "Step"
3 - - return new StepWrapper(dataFileName)
4 - else if **formatName** ← "APM-I"
5 - - return new APMInstanceWrapper(dataFileName)
6 - repeat 4 and 5 for each type of format supported.

Step 1 gets the name of the format registered for this source set. If **formatName** is "Step", then Step 3 returns a **StepWrapper**. If **formatName** is "APM-I", then Step 5 returns an **APMInstanceWrapper**. Additional formats may be added from then on.

W.1 ConstraintNetworkRelation Class Operations

Custom Constructor

- Signature:

  ConstraintNetworkRelation(String **namePrefix** , String **relationName** , ConstraintNetwork **network** , String **expression** , String **category**)

- Local Variables:

  ListOfStrings **variableNames**

  String **prefixedTempVariableName**

  ConstraintNetworkVariable **constraintNetworkVariable**

- Procedure:

1 - **variableNames** ← parse **expression** extracting the names of the variables participating in the relation.
2 - for each String **tempVariableName** ∈ **variableNames**:
3 - - **prefixedTempVariableName** ← **namePrefix** + "." + **tempVariableName**
4 - - **constraintNetworkVariable** ← **this**.**constraintNetwork**.getVariable(**prefixedVariableName**)
5 - - **this**.addVariable(**constraintNetworkVariable**)
6 - - **constraintNetworkVariable**.addRelation(**this**)

Step 1 parses **expression** extracting the names of the variables participating in the relation. Step 2 gets one name from **variableNames**. Step 3 prefixes it with **namePrefix** and a dot. Step 4 uses operation **ConstraintNetwork.getVariable** to find if a **ConstraintNetworkVariable** with **prefixedVariableName** already exists in the constraint network. If not, creates a new one and adds it to the network. Step 5 adds **constraintNetworkVariable** to the list of connected variables of this **ConstraintNetworkRelation**. Step 6 adds this **ConstraintNetworkRelation** to the list of connected relations of **constraintNetworkVariable**.

APPENDIX I

JAVA IMPLEMENTATION ONLINE DOCUMENTATION

The complete list of classes, variables and methods that form the Java prototype implementation of this thesis is available on the Internet at (Tamburini 1999). These web pages were created directly from the Java source code using Sun Microsystem's documentation-generation utility Javadoc (Flanagan 1997; Sun Microsystems 1998). The pages also include links to the complete source code of each class, which was translated into HTML using a utility called Java Beautifuller (Masquelier 1996).

The specification of each class of the APM Protocol is roughly presented as follows:

1. *Name of the Class.*
2. *Inheritance Path:* a diagram of the inheritance path of the class showing its parent classes up to the root class (in the case of Java, the root class is java.lang.Object).
3. *Class Declaration:* indicating if the class is abstract. Abstract classes cannot be instantiated (they normally have subtypes than can), and are mainly used to group variables and methods that are common to a group of classes.
4. *Link to source code:* displays the source code of the class.
5. *Variables:* lists the variables of the class, also known as *class variables*. Class variables are specified in the following form:

   ```
   [ access ] type variable_name
   ```

   Where ***access*** indicates what methods can access the value of the variable. Access can be private, protected and public. A *private* variable can only be accessed by methods of the class. A *protected* variable can be accessed by methods of the class and its subtypes. A *public* variable can be accessed by methods of any class. The ***type*** of the

variable may be a primitive data type (double, string , integer, etc.) or a class. ***variable_name*** is the name of the variable.

6. *Constructors:* lists all the combinations of constructors available for the class. A constructor is a special operation used to create new instances of the class. Constructors have the same name as the class, and are invoked automatically each time an object of the class is instantiated. It is common to have several constructors for a given class, providing a variety of means for initializing objects of the class. When an object of a class is created, initializers can be provided as arguments to the constructor. These initializers may be used to initialize the class variables of the instance being created. Which parameters are assigned to which class variables is specified in the "**Parameters**" section included for each constructor. A constructor may also perform checks on the values being used to initialize the class variables. In such cases, a brief explanation is provided under the "**Returns**" section.
7. *Methods (Operations):* lists the operations of each class (also known as *class methods*). These are operations that can be performed on instances of the class and may access the values of the class variables and public variables of other classes. In this specification, the *signature* and the *return type* of each operation is defined. The signature of an operation is the combination of the operation's name and its parameter types. The return type is the type of the value returned by the operation. In addition, a short explanation of what the operation does or returns is provided under a section labeled "**Returns**". When an operation is overriding another operation defined in a parent class, an additional section labeled "**Overrides**" specifies the name of the parent class whose operation is being overridden.

   In general, operations are specified as follows (brackets indicate optional terms):

```
[access] [abstract] return_type operation_name( [ parameter_list ] )
```

   Where ***access*** can be private, public and protected. A *private* operation is accessible only to operations of the class. A *protected* operation is accessible to operations of the class and its subtypes. A *public* operation is accessible to operations of any class. The

word **`abstract`** is used if the method is abstract, meaning that it *must* be implemented by all the subtypes of the class. ***return_type*** indicates the type of the value returned by the operation. If nothing is returned, **`void`** is used. ***operation_name*** is the name of the operation. The same class may have more than one operation with the same name, providing that their signatures are different (a feature of object-oriented programming languages known as *operation overloading*). ***parameter_list*** is a comma-separated list of the parameters passed as arguments to the operation and their types. This list may be empty, indicating that no parameters are passed to the operation.

APPENDIX J

TEST CASES DEFINITION FILES

Z.1 Test Cases APM Definition Files

Flap Link APM

```
APM flap_link;

SOURCE_SET flap_link_geometric_model ROOT_DOMAIN flap_link;

DOMAIN flap_link;
   ESSENTIAL part_number : STRING;
   IDEALIZED effective_length : REAL;
   sleeve_1 : sleeve;
   sleeve_2 : sleeve;
   shaft : beam;
   rib_1 : rib;
   rib_2 : rib;
   ESSENTIAL material : STRING;
 PRODUCT_RELATIONS
   pr1 : "<rib_1.length> == <sleeve_1.width>";
   pr2 : "<rib_2.length> == <sleeve_2.width>";
 PRODUCT_IDEALIZATION_RELATIONS
   pir1 : "<effective_length> == <sleeve_2.center.x> - <sleeve_1.center.x> -
           <sleeve_1.radius> - <sleeve_2.radius>";
   pir2 : "<shaft.wf> == <sleeve_1.width>";
   pir3 : "<shaft.hw> == 2*( <sleeve_1.radius> + <sleeve_1.thickness> -
           <shaft.tf> )";
   pir4 : "<shaft.length> == <effective_length> - <sleeve_1.thickness>
         - <sleeve_2.thickness>";
END_DOMAIN;

DOMAIN sleeve;
   ESSENTIAL width : REAL;
   ESSENTIAL thickness : REAL;
   ESSENTIAL radius : REAL;
   center : coordinates;
END_DOMAIN;

DOMAIN coordinates;
   ESSENTIAL x : REAL;
   ESSENTIAL y : REAL;
END_DOMAIN;

DOMAIN beam;
   critical_cross_section : MULTI_LEVEL cross_section;
   length : REAL;
   ESSENTIAL tf : REAL;
   ESSENTIAL tw : REAL;
   ESSENTIAL t2f : REAL;
   ESSENTIAL wf : REAL;
   ESSENTIAL hw : REAL;
```

```
 PRODUCT_IDEALIZATION_RELATIONS
   pir5 : "<critical_cross_section.detailed.tf> == <tf>";
   pir6 : "<critical_cross_section.detailed.tw> == <tw>";
   pir7 : "<critical_cross_section.detailed.t2f> == <t2f>";
   pir8 : "<critical_cross_section.detailed.wf> == <wf>";
   pir9 : "<critical_cross_section.detailed.hw> == <hw>";
END_DOMAIN;

MULTI_LEVEL_DOMAIN cross_section;
   detailed : detailed_I_section;
   simple : simple_I_section;
 PRODUCT_IDEALIZATION_RELATIONS
   pir10 : "<detailed.wf> == <simple.wf>";
   pir11 : "<detailed.hw> == <simple.hw>";
   pir12 : "<detailed.tf> == <simple.tf>";
   pir13 : "<detailed.tw> == <simple.tw>";
END_MULTI_LEVEL_DOMAIN;

DOMAIN simple_I_section SUBTYPE_OF I_section;
 PRODUCT_IDEALIZATION_RELATIONS
   pir14: "<area> == 2*<wf>*<tf> + <tw>*<hw>";
END_DOMAIN;

DOMAIN detailed_I_section SUBTYPE_OF I_section;
   IDEALIZED t1f : REAL;
   IDEALIZED t2f : REAL;
 PRODUCT_IDEALIZATION_RELATIONS
   pir15: "<area> == <wf>*( <t1f> + <t2f> ) + <tw>*( <t2f> - <t1f> ) +
           <tw>*<hw>";
   pir16: "<t1f> == <tf>";
END_DOMAIN;

DOMAIN I_section;
   IDEALIZED wf : REAL;
   IDEALIZED tf : REAL;
   IDEALIZED tw : REAL;
   IDEALIZED hw : REAL;
   IDEALIZED area : REAL;
END_DOMAIN;

DOMAIN rib;
   ESSENTIAL base : REAL;
   ESSENTIAL height : REAL;
   length : REAL;
END_DOMAIN;

END_SOURCE_SET;


SOURCE_SET flap_link_material_properties ROOT_DOMAIN material;

DOMAIN material;
   ESSENTIAL name : STRING;
   stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;
```

```
MULTI_LEVEL_DOMAIN material_levels;
   temperature_independent_linear_elastic : linear_elastic_model;
   temperature_dependent_linear_elastic :
      temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;

DOMAIN linear_elastic_model;
   IDEALIZED youngs_modulus : REAL;
   IDEALIZED poissons_ratio : REAL;
   IDEALIZED cte : REAL;
END_DOMAIN;

DOMAIN temperature_dependent_linear_elastic_model;
   IDEALIZED transition_temperature : REAL;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
   flap_link_geometric_model.flap_link.material ==
      flap_link_material_properties.material.name;
END_LINK_DEFINITIONS;

END_APM;
```

Back Plate APM

```
APM my_apm;

(* Back Plate Test Case APM *)

SOURCE_SET back_plate_geometric_model ROOT_DOMAIN plate;

DOMAIN part;
   part_number : STRING;
   designer : STRING;
END_DOMAIN;

DOMAIN plate SUBTYPE_OF part;
   l1 : REAL;
   l2 : REAL;
   l3 : REAL;
   ESSENTIAL width : REAL;
   ESSENTIAL length : REAL;
   ESSENTIAL thickness : REAL;
   hole1 : hole;
   hole2 : hole;
   ESSENTIAL material : STRING;
   IDEALIZED critical_area : REAL;

 PRODUCT_IDEALIZATION_RELATIONS
   pir_1 : "<critical_area> == ( <width> - <hole1.diameter> ) * <thickness>";
   pir_2 : "<hole1.center.y> == <width>/2";
   pir_3 : "<hole2.center.y> == <width>/2";
   pir_4 : "<l1> == <hole1.center.x>";
   pir_5 : "<l2> == <hole2.center.x> - <l1>";

 PRODUCT_RELATIONS
   pr_1 : "<length> == <l1> + <l2> + <l3>";

END_DOMAIN;

DOMAIN hole;
   ESSENTIAL diameter : REAL;
   area : REAL;
   center : coordinate;
 PRODUCT_RELATIONS
   pr_2: "<area> == Pi * <diameter>^2 / 4";
END_DOMAIN;

DOMAIN coordinate;
   ESSENTIAL x : REAL;
   ESSENTIAL y : REAL;
```

```
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET back_plate_material_data ROOT_DOMAIN material;

DOMAIN material;
   materialName : STRING;
   ESSENTIAL youngsModulus : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET back_plate_employee_data ROOT_DOMAIN person;

DOMAIN person;
   first_name : STRING;
   last_name  : STRING;
   ssn : STRING;
END_DOMAIN;

END_SOURCE_SET;

LINK_DEFINITIONS
   back_plate_geometric_model.plate.material ==
      back_plate_material_data.material.materialName;
   back_plate_geometric_model.part.designer ==  back_plate_employee_data.person.ssn;
END_LINK_DEFINITIONS;

END_APM;
```

Wing Flap Support APM

```
APM simple_inboard_beam;

SOURCE_SET simple_inboard_beam ROOT_DOMAIN inboard_beam;

DOMAIN inboard_beam;
   leg_1 : leg;
END_DOMAIN;

DOMAIN leg;
   cavity_3 : cavity_with_bottom_hole;
   rib_8 : rib;
   rib_9 : rib;
   bulkhead_attach_point : channel_fitting;
 PRODUCT_IDEALIZATION_RELATIONS
   pir1: "<bulkhead_attach_point.end_pad.width> == <rib_8.thickness>/2 +
          <cavity_3.inner_width> + <rib_9.thickness>/2";
```

```
   pir2: "<bulkhead_attach_point.end_pad.height> ==
          <cavity_3.bottom_thickness>/2 + <cavity_3.inner_breadth>";
   pir3: "<bulkhead_attach_point.end_pad.thickness> ==
          <cavity_3.minimum_base_thickness>";
   pir4: "<bulkhead_attach_point.end_pad.hole_diameter> ==
          <cavity_3.hole_diameter>";
   pir5: "<bulkhead_attach_point.end_pad.hole_center_height> ==
          <cavity_3.hole_height> + <cavity_3.bottom_thickness>";
   pir6: "<bulkhead_attach_point.base.width> ==
          <bulkhead_attach_point.end_pad.width>";
   pir7: "<bulkhead_attach_point.base.height> ==
          <cavity_3.inner_height> + <cavity_3.minimum_base_thickness>/2";
   pir8: "<bulkhead_attach_point.base.thickness> ==
          <cavity_3.bottom_thickness>";
   pir9: "<bulkhead_attach_point.base.hole_diameter> == 0";
   pir10: "<bulkhead_attach_point.base.hole_center_height> == 0";
   pir11: "<bulkhead_attach_point.wall.width> ==
           <bulkhead_attach_point.base.height>";
   pir12: "<bulkhead_attach_point.wall.height>
         ==<bulkhead_attach_point.end_pad.height>";
   pir13: "<bulkhead_attach_point.wall.thickness> == ( <rib_8.thickness> +
           <rib_9.thickness> )/2";
END_DOMAIN;

DOMAIN cavity_with_bottom_hole;
   inner_width : REAL;
   inner_breadth : REAL;
   inner_height : REAL;
   minimum_base_thickness : REAL;
   top_thickness : REAL;
   bottom_thickness : REAL;
   hole_diameter : REAL;
   hole_height : REAL;
END_DOMAIN;

DOMAIN rib;
   thickness : REAL;
END_DOMAIN;

DOMAIN channel_fitting;
   end_pad : wall_with_hole;
   base : wall_with_hole;
   wall : wall;
END_DOMAIN;

DOMAIN wall;
   IDEALIZED width : REAL;
   IDEALIZED height : REAL;
   IDEALIZED thickness : REAL;
END_DOMAIN;

DOMAIN wall_with_hole SUBTYPE_OF wall;
   IDEALIZED hole_diameter : REAL;
   IDEALIZED hole_center_height : REAL;
END_DOMAIN;
```

```
END_SOURCE_SET;

END_APM;
```

Printed Wiring Assembly APM

```
APM my_apm;

SOURCE_SET pwa_data ROOT_DOMAIN part;

DOMAIN part;
   part_number : STRING;
   designer : STRING;
END_DOMAIN;

DOMAIN pwa SUBTYPE_OF part;
   thickness : REAL;
   max_thickness : REAL;
   min_thickness : REAL;
   number_of_layers : REAL;
   outline : LIST[0,?] OF coordinate;
   pwb_material : STRING;
   layup : LIST[0,?] OF layer;
   materials : LIST[0,?] OF STRING;
   numbers : LIST[0,?] OF REAL;
   max_number : REAL;
   min_number : REAL;
   count_numbers : REAL;
   sum_numbers : REAL;
   alpha_sub_b : REAL;
   width : REAL;
   length : REAL;
   total_diagonal : REAL;
   test_attr : test;
 PRODUCT_RELATIONS
   pr1: "<thickness> == <layup.SUM[thickness]>";
   pr2: "<max_thickness> == <layup.MAX[thickness]>";
   pr3: "<min_thickness> == <layup.MIN[thickness]>";
   pr4: "<number_of_layers> == <layup.COUNT>";
   pr5: "<max_number> == <numbers.MAX>";
   pr6: "<min_number> == <numbers.MIN>";
   pr7: "<count_numbers> == <numbers.COUNT>";
   pr8: "<sum_numbers> == <numbers.SUM>";
 PRODUCT_IDEALIZATION_RELATIONS
   pir1: "<alpha_sub_b> == <layup.SUM[ta]>/<thickness>";
   pir2: "<width> == <outline.MAX[x]>-<outline.MIN[x]>";
   pir3: "<length> == <outline.MAX[y]>-<outline.MIN[y]>";
   pir4: "<total_diagonal>^2 == <width>^2 + <length>^2";
END_DOMAIN;

DOMAIN test;
   t1 : LIST[0,?] OF B;
```

```
   t2 : STRING;
   t3 : REAL;
 PRODUCT_RELATIONS
   pr1 : "<t3> == <t1.SUM[b3]>";
END_DOMAIN;

DOMAIN B;
   b1 : REAL;
   b2 : STRING;
   b3 : REAL;
   b4 : LIST[0,?] OF REAL;
 PRODUCT_RELATIONS
   pr1 : "<b3> == <b4.SUM>";
END_DOMAIN;

DOMAIN layer;
   material : STRING;
   thickness : REAL;
   ta : REAL;
 PRODUCT_RELATIONS
   pr1: "<ta> == <thickness>*<material.stress_strain_model.
         temperature_independent_linear_elastic.cte>";
END_DOMAIN;

DOMAIN coordinate;
   x : REAL;
   y : REAL;
END_DOMAIN;

END_SOURCE_SET;

SOURCE_SET layer_material_data ROOT_DOMAIN material;

DOMAIN material;
   name : STRING;
   stress_strain_model : MULTI_LEVEL material_levels;
END_DOMAIN;

MULTI_LEVEL_DOMAIN material_levels;
   temperature_independent_linear_elastic : linear_elastic_model;
   temperature_dependent_linear_elastic :
      temperature_dependent_linear_elastic_model;
END_MULTI_LEVEL_DOMAIN;

DOMAIN linear_elastic_model;
   youngs_modulus : REAL;
   poissons_ratio : REAL;
   cte : REAL;
END_DOMAIN;

DOMAIN temperature_dependent_linear_elastic_model;
END_DOMAIN;

END_SOURCE_SET;
```

```
LINK_DEFINITIONS
 pwa_data.pwa.pwb_material == layer_material_data.material.name;
 pwa_data.pwa.layup.layer.material == layer_material_data.material.name;
 pwa_data.pwa.test_attr.t1.B.b2 == layer_material_data.material.name;
END_LINK_DEFINITIONS;

END_APM;
```

DD.1 Test Cases Design Data Files

Flap Link Test Case

APM-I Format (source set flap_link_geometric_model)

```
DATA;

INSTANCE_OF flap_link;
   part_number : "FLAP-001";
   effective_length : 12.5;
   sleeve_1.width : 1.5;
   sleeve_1.thickness : 0.5;
   sleeve_1.radius : 0.5;
   sleeve_1.center.x : 0.0;
   sleeve_1.center.y : 0.0;
   sleeve_2.width : 2.0;
   sleeve_2.thickness : 0.6;
   sleeve_2.radius : 0.75;
   sleeve_2.center.x : ?;
   sleeve_2.center.y : 0.0;
   shaft.length : ?;
   shaft.tf : 0.1;
   shaft.tw : 0.1;
   shaft.t2f : 0.15;
   shaft.wf : ?;
   shaft.hw : ?;
   shaft.critical_cross_section.detailed.wf : ?;
   shaft.critical_cross_section.detailed.tf : ?;
   shaft.critical_cross_section.detailed.tw : ?;
   shaft.critical_cross_section.detailed.hw : ?;
   shaft.critical_cross_section.detailed.area : ?;
   shaft.critical_cross_section.detailed.t1f : ?;
   shaft.critical_cross_section.detailed.t2f : ?;
   shaft.critical_cross_section.simple.wf : ?;
   shaft.critical_cross_section.simple.tf : ?;
   shaft.critical_cross_section.simple.tw : ?;
   shaft.critical_cross_section.simple.hw : ?;
   shaft.critical_cross_section.simple.area : ?;
   rib_1.base : 10.00;
   rib_1.height : 0.5;
   rib_1.length : ?;
   rib_2.base : 10.00;
   rib_2.height : 0.5;
   rib_2.length : ?;
   material : "aluminum";
END_INSTANCE;


INSTANCE_OF flap_link;
   part_number : "FLAP-002";
```

```
   effective_length : ?;
   sleeve_1.width : 1.5;
   sleeve_1.thickness : 0.5;
   sleeve_1.radius : 0.5;
   sleeve_1.center.x : 0.0;
   sleeve_1.center.y : 0.0;
   sleeve_2.width : 2.0;
   sleeve_2.thickness : 0.6;
   sleeve_2.radius : 0.75;
   sleeve_2.center.x : 20.00;
   sleeve_2.center.y : 0.0;
   shaft.length : ?;
   shaft.tf : 0.1;
   shaft.tw : 0.1;
   shaft.t2f : 0.15;
   shaft.wf : ?;
   shaft.hw : ?;
   shaft.critical_cross_section.detailed.wf : ?;
   shaft.critical_cross_section.detailed.tf : ?;
   shaft.critical_cross_section.detailed.tw : ?;
   shaft.critical_cross_section.detailed.hw : ?;
   shaft.critical_cross_section.detailed.area : ?;
   shaft.critical_cross_section.detailed.t1f : ?;
   shaft.critical_cross_section.detailed.t2f : ?;
   shaft.critical_cross_section.simple.wf : ?;
   shaft.critical_cross_section.simple.tf : ?;
   shaft.critical_cross_section.simple.tw : ?;
   shaft.critical_cross_section.simple.hw : ?;
   shaft.critical_cross_section.simple.area : ?;
   rib_1.base : 10.00;
   rib_1.height : 0.5;
   rib_1.length : ?;
   rib_2.base : 10.00;
   rib_2.height : 0.5;
   rib_2.length : ?;
   material : "steel";
END_INSTANCE;

END_DATA;
```

APM-I Format (source set flap_link_material_properties)

```
DATA;

INSTANCE_OF material;
   name : "steel";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      30000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.30;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.0000065;
   stress_strain_model.temperature_dependent_linear_elastic.transition_temperature :
      275.00;
END_INSTANCE;
```

```
INSTANCE_OF material;
   name : "aluminum";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      10400000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.000013;
   stress_strain_model.temperature_dependent_linear_elastic.transition_temperature :
      156.00;
END_INSTANCE;

INSTANCE_OF material;
   name : "cast iron";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      18000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.000006;
   stress_strain_model.temperature_dependent_linear_elastic.transition_temperature :
      125.00;
END_INSTANCE;

END_DATA;
```

STEP Format (source set flap_link_geometric_model)

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER 1.6 */

FILE_DESCRIPTION(
/* description */ (''),
/* implementation_level */ '2;1');

FILE_NAME(
/* name */ 'flap_link_geometric_data',
/* time_stamp */ '1998-07-22T14:44:30-04:00',
/* author */ (''),
/* organization */ (''),
/* preprocessor_version */ 'ST-DEVELOPER 1.6',
/* originating_system */ '',
/* authorisation */ '');

FILE_SCHEMA (('FLAP_LINK_GEOMETRIC_MODEL'));
ENDSEC;

DATA;
#10=FLAP_LINK('FLAP-001',12.5,#20,#40,#60,#100,#110,'aluminum');
#20=SLEEVE(1.5,0.5,0.5,#30);
```

```
#30=COORDINATES(0.0,0.0);
#40=SLEEVE(2.0,0.6,0.75,#50);
#50=COORDINATES(-999.00,0.0);
#60=BEAM(#70,-999.00,0.1,0.1,0.15,-999.00,-999.00);
#70=CROSS_SECTION(#80,#90);
#80=DETAILED_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00,-999.00,-999.00);
#90=SIMPLE_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00);
#100=RIB(10.0,0.5,-999.00);
#110=RIB(10.0,0.5,-999.00);

#120=FLAP_LINK('FLAP-002',-999.00,#130,#150,#170,#210,#220,'steel');
#130=SLEEVE(1.5,0.5,0.5,#140);
#140=COORDINATES(0.0,0.0);
#150=SLEEVE(2.0,0.6,0.75,#160);
#160=COORDINATES(20.00,0.0);
#170=BEAM(#180,-999.00,0.1,0.1,0.15,-999.00,-999.00);
#180=CROSS_SECTION(#190,#200);
#190=DETAILED_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00,-999.00,-999.00);
#200=SIMPLE_I_SECTION(-999.00,-999.00,-999.00,-999.00,-999.00);
#210=RIB(10.0,0.5,-999.00);
#220=RIB(10.0,0.5,-999.00);

ENDSEC;
END-ISO-10303-21;
```

STEP Format (source set flap_link_material_properties)

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER 1.6 */

FILE_DESCRIPTION(
/* description */ (''),
/* implementation_level */ '2;1');

FILE_NAME(
/* name */ 'flap_link_material_properties_data',
/* time_stamp */ '1998-08-31T10:24:16-04:00',
/* author */ (''),
/* organization */ (''),
/* preprocessor_version */ 'ST-DEVELOPER 1.6',
/* originating_system */ '',
/* authorisation */ '');

FILE_SCHEMA (('FLAP_LINK_MATERIAL_PROPERTIES'));
ENDSEC;

DATA;
#10=MATERIAL('steel',#11);
#11=MATERIAL_LEVELS(#12,#13);
#12=LINEAR_ELASTIC_MODEL(30000000.00,0.30,0.0000065);
#13=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(275.00);
```

```
#110=MATERIAL('aluminum',#111);
#111=MATERIAL_LEVELS(#112,#113);
#112=LINEAR_ELASTIC_MODEL(10400000.00,0.25,0.000013);
#113=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(156.00);

#210=MATERIAL('cast iron',#211);
#211=MATERIAL_LEVELS(#212,#213);
#212=LINEAR_ELASTIC_MODEL(18000000.00,0.25,0.000006);
#213=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL(125.00);


ENDSEC;
END-ISO-10303-21;
```

Back Plate Test Case

APM-I Format (source set back_plate_geometric_model)

```
DATA;

INSTANCE_OF plate;
part_number : "XYZ-001";
designer : "1234";
l1 : ?;
l2 : ?;
l3 : 5.00;
width : 20.00;
length : ?;
thickness : 0.25;
hole1.diameter : ?;
hole1.center.x : 10.00;
hole1.center.y : ?;
hole1.area : ?;
hole2.diameter : 6.00;
hole2.center.x : 20.00;
hole2.center.y : ?;
hole2.area : ?;
material : "steel";
critical_area : ?;
END_INSTANCE;

INSTANCE_OF plate;
part_number : "XYZ-002";
designer : "567";
l1 : ?;
l2 : ?;
l3 : ?;
width : 25.00;
length : 35.00;
thickness : 0.30;
hole1.diameter : 9.00;
hole1.center.x : 12.00;
hole1.center.y : ?;
hole1.area : ?;
hole2.diameter : 6.00;
hole2.center.x : 20.00;
hole2.center.y : ?;
hole2.area : ?;
material : "aluminum";
critical_area : ?;
```

```
END_INSTANCE;

INSTANCE_OF part;
part_number : "XYZ-001";
designer : "567";
END_INSTANCE;

END_DATA;
```

APM-I Format (source set back_plate_material_data)

```
DATA;

INSTANCE_OF material;
   materialName : "steel";
   youngsModulus : 3000000.00;
END_INSTANCE;

INSTANCE_OF material;
   materialName : "aluminum";
   youngsModulus : 10400000.00;
END_INSTANCE;

END_DATA;
```

APM-I Format (source set back_plate_employee_data)

```
DATA;

INSTANCE_OF person;
   first_name : "Diego";
   last_name  : "Tamburini";
   ssn : "1234";
END_INSTANCE;

INSTANCE_OF person;
   first_name : "Patricia";
   last_name  : "Esparza";
   ssn : "567";
END_INSTANCE;

END_DATA;
```

STEP Format (source set back_plate_geometric_model)

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER 1.6 */

FILE_DESCRIPTION(
/* description */ (''),
/* implementation_level */ '2;1');

FILE_NAME(
/* name */ 'back_plate_geometric_data',
/* time_stamp */ '1998-06-03T10:40:09-04:00',
/* author */ (''),
/* organization */ (''),
/* preprocessor_version */ 'ST-DEVELOPER 1.6',
/* originating_system */ '',
/* authorisation */ '');

FILE_SCHEMA (('BACK_PLATE_GEOMETRIC_MODEL'));
ENDSEC;

DATA;
#10=PLATE('XYZ-001','1234',-999.00,-999.00,5.00,20.00,-999.00,0.25,#20,#40,'steel',
          -999.00);
#20=HOLE(-999.00,-999.00,#30);
#30=COORDINATE(10.0,-999.00);
#40=HOLE(6.00,-999.00,#50);
#50=COORDINATE(20.0,-999.00);

#100=PLATE('XYZ-002','567',-999.00,-999.00,-999.00,25.00,35.00,0.30,#200,#40,'aluminum',
           -999.00);
#200=HOLE(9.0,-999.00,#300);
#300=COORDINATE(12.0,-999.00);
#400=HOLE(6.00,-999.00,#500);
#500=COORDINATE(20.0,-999.00);


ENDSEC;
END-ISO-10303-21;
```

Wing Flap Support Test Case

APM-I Format (source set simple_inboard_beam)

```
DATA;

INSTANCE_OF inboard_beam;

   leg_1.cavity_3.inner_width : 2.13;
   leg_1.cavity_3.inner_breadth : 1.9345;
   leg_1.cavity_3.inner_height : 2.5932;
   leg_1.cavity_3.minimum_base_thickness : 0.50;
   leg_1.cavity_3.top_thickness : 0.45;
   leg_1.cavity_3.bottom_thickness : 0.307;
   leg_1.cavity_3.hole_diameter : 0.875;
   leg_1.cavity_3.hole_height : 0.96;

   leg_1.rib_8.thickness : 0.31;
   leg_1.rib_9.thickness : 0.31;

   leg_1.bulkhead_attach_point.end_pad.width : ?;
   leg_1.bulkhead_attach_point.end_pad.height : ?;
   leg_1.bulkhead_attach_point.end_pad.thickness : ?;
   leg_1.bulkhead_attach_point.end_pad.hole_diameter : ?;
   leg_1.bulkhead_attach_point.end_pad.hole_center_height : ?;

   leg_1.bulkhead_attach_point.base.width : ?;
   leg_1.bulkhead_attach_point.base.height : ?;
   leg_1.bulkhead_attach_point.base.thickness : ?;
   leg_1.bulkhead_attach_point.base.hole_diameter : ?;
   leg_1.bulkhead_attach_point.base.hole_center_height : ?;

   leg_1.bulkhead_attach_point.wall.width : ?;
   leg_1.bulkhead_attach_point.wall.height : ?;
   leg_1.bulkhead_attach_point.wall.thickness : ?;

END_INSTANCE;

END_DATA;
```

STEP Format

Not Available

Printed Wiring Assembly Test Case

APM-I Format (source set pwa_data)

```
DATA;

INSTANCE_OF pwa;
   part_number : "PWA-123" ;
   designer : "Diego Tamburini";
   thickness : ?;
   max_thickness : ?;
   min_thickness : ?;
   number_of_layers : ?;
   outline[0].x : 0.0;
   outline[0].y : 1.0 ;
   outline[1].x : 0.0;
   outline[1].y : 5.7 ;
   outline[2].x : 10.0;
   outline[2].y : 5.7 ;
   outline[3].x : 7.0;
   outline[3].y : 0.0 ;
   outline[4].x : 3.0;
   outline[4].y : 0.0 ;
   pwb_material : "epoxy";
   layup[0].material : "copper";
   layup[0].thickness : 0.2;
   layup[0].ta : ?;
   layup[1].material : "FR5";
   layup[1].thickness : 0.5;
   layup[1].ta : ?;
   layup[2].material : "steel";
   layup[2].thickness : 0.7;
   layup[2].ta : ?;
   layup[3].material : "FR5";
   layup[3].thickness : 0.1;
   layup[3].ta : ?;
   numbers[0] : 2.5;
   numbers[0] : 3.5;
   numbers[0] : 1.1;
   numbers[0] : 7.5;
   numbers[0] : -1.5;
   materials[0]: "steel";
   materials[1]: "epoxy";
   materials[2]: "aluminum";
   max_number : ?;
```

```
   min_number : ?;
   count_numbers : ?;
   sum_numbers : ?;
   alpha_sub_b : ?;
   width : ?;
   length : ?;
   total_diagonal : ?;
   test_attr.t1[0].b1 : 2.2 ;
   test_attr.t1[0].b2 : "steel";
   test_attr.t1[0].b3 : ? ;
   test_attr.t1[0].b4[0] : 1.0 ;
   test_attr.t1[0].b4[1] : 2.0 ;
   test_attr.t1[0].b4[2] : 3.0 ;
   test_attr.t1[1].b1 : 3.6 ;
   test_attr.t1[1].b2 : "aluminum";
   test_attr.t1[1].b3 : ? ;
   test_attr.t1[1].b4[0] : 4.0 ;
   test_attr.t1[1].b4[1] : 5.0 ;
   test_attr.t1[1].b4[2] : 6.0 ;
   test_attr.t1[2].b1 : 1.1 ;
   test_attr.t1[2].b2 : "FR5";
   test_attr.t1[2].b3 : ? ;
   test_attr.t1[2].b4[0] : 7.0 ;
   test_attr.t1[2].b4[1] : 8.0 ;
   test_attr.t1[2].b4[2] : 9.0 ;
   test_attr.t2 : "Hello alone";
   test_attr.t3 : ?;
END_INSTANCE;

END_DATA;
```

APM-I Format (source set layer_material_data)

```
DATA;

INSTANCE_OF material;
   name : "steel";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      30000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.00001;
END_INSTANCE;

INSTANCE_OF material;
   name : "aluminum";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      15000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.25;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.00002;
END_INSTANCE;

INSTANCE_OF material;
```

```
   name : "cast_iron";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      27000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.15;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.00098;
END_INSTANCE;

INSTANCE_OF material;
   name : "copper";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      29000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.15;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.00098;
END_INSTANCE;

INSTANCE_OF material;
   name : "epoxy";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      16000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.2;
   stress_strain_model.temperature_independent_linear_elastic.cte : 0.75;
END_INSTANCE;

INSTANCE_OF material;
   name : "FR5";
   stress_strain_model.temperature_independent_linear_elastic.youngs_modulus :
      11000000.00;
   stress_strain_model.temperature_independent_linear_elastic.poissons_ratio : 0.15;
   stress_strain_model.temperature_independent_linear_elastic.cte : 5.0;
END_INSTANCE;

END_DATA;
```

STEP Format (source set pwa_data)

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER 1.6 */

FILE_DESCRIPTION(
/* description */ (''),
/* implementation_level */ '2;1');

FILE_NAME(
/* name */ 'pwa_data',
/* time_stamp */ '1998-06-01T10:36:46-04:00',
/* author */ ('Diego R. Tamburini'),
/* organization */ ('EIS Lab - Georgia Institute of Technology'),
```

```
/* preprocessor_version */ 'ST-DEVELOPER 1.6',
/* originating_system */ '',
/* authorisation */ '');

FILE_SCHEMA (('PWA_DATA'));
ENDSEC;

DATA;
#10=PWA('PWA-123','Diego Tamburini',-999.00,-999.00,-999.00,
         -999.00,(#12,#13,#14,#15,#16),'epoxy',(#20,#21,#22,#23),
         ('steel','epoxy','aluminum'),(2.5 , 3.5 , 1.1 , 7.5 , -1.5),-999.00,-999.00,
         -999.00,-999.00,-999.00,-999.00,-999.00,-999.00,#100);
#12=COORDINATE(0.0,1.0);
#13=COORDINATE(0.0,5.7);
#14=COORDINATE(10.0,5.7);
#15=COORDINATE(7.0,0.0);
#16=COORDINATE(3.0,0.0);
#20=LAYER('copper',0.2,-999.00);
#21=LAYER('FR5',0.5,-999.00);
#22=LAYER('steel',0.7,-999.00);
#23=LAYER('FR5',0.1,-999.00);
#100=TEST( (#110,#120,#130), 'test_att t2' ,-999.00);
#110=B(110.00,'steel',-999.00, (1.0,2.0,3.0) );
#120=B(120.00,'aluminum',-999.00, (4.0,5.0,6.0) );
#130=B(130.00,'FR5',-999.00, (7.0,8.0,9.0) );

#200=PART('Foo-999','Foo Guy');
ENDSEC;
END-ISO-10303-21;
```

STEP Format (source set layer_material_data)

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER 1.6 */

FILE_DESCRIPTION(
/* description */ (''),
/* implementation_level */ '2;1');

FILE_NAME(
/* name */ 'pwa_material_data',
/* time_stamp */ '1998-06-18T08:44:02-04:00',
/* author */ (''),
/* organization */ (''),
/* preprocessor_version */ 'ST-DEVELOPER 1.6',
/* originating_system */ '',
/* authorisation */ '');

FILE_SCHEMA (('LAYER_MATERIAL_DATA'));
ENDSEC;

DATA;
```

```
#10=MATERIAL( 'steel' , #11 );
#11=MATERIAL_LEVELS( #12 , #13 );
#12=LINEAR_ELASTIC_MODEL( 30000000.00 , 0.25 , 0.00001 );
#13=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

#20=MATERIAL( 'aluminum' , #21 );
#21=MATERIAL_LEVELS( #22 , #23 );
#22=LINEAR_ELASTIC_MODEL( 15000000.00 , 0.25 , 0.00002 );
#23=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

#30=MATERIAL( 'cast_iron' , #31 );
#31=MATERIAL_LEVELS( #32 , #33 );
#32=LINEAR_ELASTIC_MODEL( 27000000.00 , 0.15 , 0.00098 );
#33=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

#40=MATERIAL( 'copper' , #41 );
#41=MATERIAL_LEVELS( #42 , #43 );
#42=LINEAR_ELASTIC_MODEL( 29000000.00 , 0.15 , 0.00098 );
#43=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

#50=MATERIAL( 'epoxy' , #51 );
#51=MATERIAL_LEVELS( #52 , #53 );
#52=LINEAR_ELASTIC_MODEL( 16000000.00 , 0.2 , 0.75 );
#53=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

#60=MATERIAL( 'FR5' , #61 );
#61=MATERIAL_LEVELS( #62 , #63 );
#62=LINEAR_ELASTIC_MODEL( 11000000.00 , 0.15 , 5.0 );
#63=TEMPERATURE_DEPENDENT_LINEAR_ELASTIC_MODEL();

ENDSEC;
END-ISO-10303-21;
```

APPENDIX K

TEST APM CLIENT APPLICATIONS CODE

II.1 PWB Bending Analysis Application

PWBBendingAnalysis.java

```
import gui.CloseWindowAndExit;
import apm.*;
import file.*;
import java.util.*;

public class PWBBendingAnalysis
{
    public static void main( String args[] )
    {
        MainFrame f = new MainFrame();
        f.addWindowListener( new CloseWindowAndExit() );
    }

}
```

MainFrame.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;
import java.util.*;
import java.io.*;
import file.*;
import gui.*;


public class MainFrame extends Frame implements ActionListener { private MenuBar bar;
private Menu fileMenu;
private MenuItem loadAPMDefinition;
private MenuItem loadData;
private MenuItem exitProgram;
private String lastDirectoryName;
private String apmDefinitionFileName;

Panel analysisPanel;


ListOfAPMComplexDomainInstances listOfPWAInstances;
APMObjectDomainInstance thePWA;


public MainFrame( )
{
    super( "PWB Bending Analysis" );


    setSize( 500 , 530 );


    // Create file menu
    bar = new MenuBar();
    fileMenu = new Menu( "File" );
    loadAPMDefinition = new MenuItem( "Load APM Definition" );
    loadData = new MenuItem( "Load PWA Data" );
    fileMenu.add( loadAPMDefinition );
    fileMenu.add( loadData );
    fileMenu.addSeparator( );
    exitProgram = new MenuItem( "Exit" );
    fileMenu.add( exitProgram );

    // Add listener to the menu items
    loadAPMDefinition.addActionListener( this );
```

```
        loadData.addActionListener( this );
        exitProgram.addActionListener( this );


        // Add menus to the bar
        bar.add( fileMenu );

        // Add bar
        setMenuBar( bar );

        setBackground( Color.lightGray );


        setVisible( true );


    }

    public void actionPerformed( ActionEvent e )
    {
        boolean success = false;
        FileDialog fileDialog;
        InfoDialog infoDialog;

        if( e.getSource() == loadAPMDefinition )
        {
            // Create file dialog and get the file name
            fileDialog = new FileDialog( this , "Select APM Definition File" , FileDialog.LOAD );
            fileDialog.show();
            lastDirectoryName = fileDialog.getDirectory();
            apmDefinitionFileName = lastDirectoryName + fileDialog.getFile();

            // Initialize the APMInterface
            APMInterface.initialize();

            success = APMInterface.loadAPMDefinitions( apmDefinitionFileName );

            // Display a dialog notifying wheter or not the APM definitions have been loaded
            if( success )
                infoDialog = new InfoDialog( this , "Message" , "APM loaded successfully" );
            else
                infoDialog = new InfoDialog( this , "Message" , "APM not loaded" );

            infoDialog.show();
        }


        else if( e.getSource() == loadData )
        {
            APMSourceSet sourceSetCursor;
            ListOfStrings listOfFileNames = new ListOfStrings();

            // Prompt user for a data file for each source set
            for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
            {
                sourceSetCursor = APMInterface.getSourceSets().elementAt( i );

                fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" , FileDialog.LOAD );
                fileDialog.setDirectory( lastDirectoryName );
                fileDialog.show();
                listOfFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
            }

            // Load the data
            success = APMInterface.loadSourceSetData( listOfFileNames );

            // Display a message indicating wheter or not the data was loaded succesfully
            if( success )
```

```
        {
            infoDialog = new InfoDialog( this , "Message" , "Data loaded successfully" );

            // Get the instances of flap_link
            listOfPWAInstances = APMInterface.getInstancesOf( "pwa" );

            // Set the default flap link instance to the first in the list of flap links
            thePWA = (APMObjectDomainInstance) listOfPWAInstances.elementAt( 0 );

            // Create the panel with PWA information and bending analysis
            analysisPanel = new AnalysisPanel( thePWA );

            // Add the analysisPanel to the frame
            add( analysisPanel , BorderLayout.NORTH );


            setVisible( true );

        }
        else
            infoDialog = new InfoDialog( this , "Message" , "Data not loaded" );

            infoDialog.show();


    }

    else if( e.getSource() == exitProgram )
    {
        System.exit( 0 );
    }
}


}
```

AnalysisPanel.java

```
import java.awt.*;
import apm.*;
import java.awt.event.*;

public class AnalysisPanel extends Panel implements ActionListener
{
    private Label pwaDescriptionLabel,
    pwaPartNumberLabel,
    pwbPartNumberLabel,
    pwbNestedThicknessLabel,
    pwbWidthLabel,
    pwbLengthLabel,
    coefficientOfThermalBendingLabel,
    deltaTLabel,
    deltaLLabel;

    private TextField pwaDescriptionField,
    pwaPartNumberField,
    pwbPartNumberField ,
    pwbNestedThicknessField,
    pwbWidthField,
    pwbLengthField,
    coefficientOfThermalBendingField,
    deltaTField,
    deltaLField;

    private Button calculateButton;
```

```
private GridBagLayout gbLayout;
private GridBagConstraints gbConstraints;


public AnalysisPanel( APMObjectDomainInstance pwaInstance )
{
    gbLayout = new GridBagLayout();
    setLayout( gbLayout );

    gbConstraints = new GridBagConstraints();
    gbConstraints.anchor = GridBagConstraints.WEST;

    pwaDescriptionLabel = new Label( "Description" );
    addComponent( pwaDescriptionLabel , 0 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwaDescriptionField = new TextField( 20 );
    pwaDescriptionField.setEditable( false );
    pwaDescriptionField.setBackground( Color.lightGray );
    addComponent( pwaDescriptionField , 0 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

    pwaPartNumberLabel = new Label( "PWA Part #" );
    addComponent( pwaPartNumberLabel , 1 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwaPartNumberField = new TextField( 20 );
    pwaPartNumberField.setEditable( false );
    pwaPartNumberField.setBackground( Color.lightGray );
    addComponent( pwaPartNumberField , 1 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

    pwbPartNumberLabel = new Label( "PWB Part #" );
    addComponent( pwbPartNumberLabel , 2 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwbPartNumberField = new TextField( 20 );
    pwbPartNumberField.setEditable( false );
    pwbPartNumberField.setBackground( Color.lightGray );
    addComponent( pwbPartNumberField , 2 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );


    pwbNestedThicknessLabel = new Label( "PWB Nested Thickness" );
    addComponent( pwbNestedThicknessLabel , 3 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwbNestedThicknessField = new TextField( 20 );
    pwbNestedThicknessField.setEditable( false );
    pwbNestedThicknessField.setBackground( Color.lightGray );
    addComponent( pwbNestedThicknessField , 3 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );


    pwbWidthLabel = new Label( "PWB Width" );
    addComponent( pwbWidthLabel , 4 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwbWidthField = new TextField( 20 );
    pwbWidthField.setEditable( false );
    pwbWidthField.setBackground( Color.lightGray );
    addComponent( pwbWidthField , 4 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

    pwbLengthLabel = new Label( "PWB Length" );
    addComponent( pwbLengthLabel , 5 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    pwbLengthField = new TextField( 20 );
    pwbLengthField.setEditable( false );
    pwbLengthField.setBackground( Color.lightGray );
    addComponent( pwbLengthField , 5 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );


    coefficientOfThermalBendingLabel = new Label( "PWB Alpha Sub B" );
    addComponent( coefficientOfThermalBendingLabel , 6 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    coefficientOfThermalBendingField = new TextField( 20 );
    coefficientOfThermalBendingField.setEditable( false );
    coefficientOfThermalBendingField.setBackground( Color.lightGray );
    addComponent( coefficientOfThermalBendingField , 6 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

    deltaTLabel = new Label( "Delta T" );
    addComponent( deltaTLabel , 7 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
    deltaTField = new TextField( 20 );
```

```
        deltaTField.setEditable( true );
        deltaTField.setBackground( Color.lightGray );
        addComponent( deltaTField , 7 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );
        deltaTField.addActionListener( this );

        deltaLLabel = new Label( "Delta L" );
        addComponent( deltaLLabel , 8 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        deltaLField = new TextField( 10 );
        deltaLField.setEditable( false );
        addComponent( deltaLField , 8 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );


        // Create a calculate button
        calculateButton = new Button( "Calculate PWB Bending" );
        calculateButton.addActionListener( this );
        addComponent( calculateButton , 9 , 0 , 1 , 1 , 20 , 40 , 20 , 40);

        deltaTField.setText( "0" );

        updateValues( pwaInstance );
        setVisible( true );
    }

    public void updateValues( APMObjectDomainInstance pwaInstance )
    {
        String pwaDescription = pwaInstance.
        getStringInstance( "description" ).
        getStringValue();

        pwaDescriptionField.setText( pwaDescription );


        String pwaPartNumber = pwaInstance.
        getStringInstance( "part_number" ).
        getStringValue();

        pwaPartNumberField.setText( pwaPartNumber );


        String pwbPartNumber = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getStringInstance( "part_number" ).
        getStringValue();

        pwbPartNumberField.setText( pwbPartNumber );


        double pwbNestedThickness = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getRealInstance( "nested_thickness" ).
        getRealValue();

        pwbNestedThicknessField.setText( Double.toString( pwbNestedThickness ) );


        double pwbWidth = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getRealInstance( "width" ).
        getRealValue();

        pwbWidthField.setText( Double.toString( pwbWidth ) );


        double pwbLength = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getRealInstance( "length" ).
        getRealValue();
```

```
        pwbLengthField.setText( Double.toString( pwbLength ) );


        double coefficientOfThermalBending = pwaInstance.
        getObjectInstance( "associated_pwb" ).
        getRealInstance( "coefficient_of_thermal_bending" ).
        getRealValue();

        coefficientOfThermalBendingField.setText( Double.toString( coefficientOfThermalBending ) );


        deltaLField.setText( "0" );



    }

    private void addComponent( Component c , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        gbConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        gbConstraints.gridx = column;
        gbConstraints.gridy = row;

        // Set gridwidth and gridheight
        gbConstraints.gridwidth = width;
        gbConstraints.gridheight = height;

        // Set constraints
        gbLayout.setConstraints( c , gbConstraints );
        add( c );
    }


    public void actionPerformed( ActionEvent e )
    {
        double alpha_sub_b = Double.valueOf( coefficientOfThermalBendingField.getText() ).doubleValue();
        double width = Double.valueOf( pwbWidthField.getText() ).doubleValue();
        double length = Double.valueOf( pwbLengthField.getText() ).doubleValue();
        double pwbNestedThickness = Double.valueOf( pwbNestedThicknessField.getText() ).doubleValue();
        double deltaT = Double.valueOf( deltaTField.getText() ).doubleValue();


        double deltaL = ( alpha_sub_b * (width*width + length*length) * deltaT )/ pwbNestedThickness;


        deltaLField.setText( Double.toString( deltaL ) );


    }
}
```

II.2 Flap Link Extension Analysis Application

FlapLinkExtensionalAnalysis.java

```
import gui.CloseWindowAndExit;
import apm.*;
import file.*;
import java.util.*;

public class FlapLinkExtensionalAnalysis
{
    public static void main( String args[] )
    {
        MainFrame f = new MainFrame();
        f.addWindowListener( new CloseWindowAndExit() );
    }
}
```

MainFrame.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;
import java.util.*;
import java.io.*;
import file.*;
import gui.*;


public class MainFrame extends Frame implements ActionListener
{
    private MenuBar bar;
    private Menu fileMenu;
    private MenuItem loadAPMDefinition;
    private MenuItem loadData;
    private MenuItem exitProgram;
    Panel topPanel , bottomPanel;
    private Canvas emptyCanvas;
    CardLayout cardManager; // Package access
    FormulaBasedAnalysisPanel formulaBasedAnalysisPanel;
    FEABasedAnalysisPanel feaBasedAnalysisPanel;
    InfoDisplayPanel infoDisplayPanel;
    private BufferedWriter stFile;
    private String lastDirectoryName;
    private String apmDefinitionFileName;

    ListOfAPMComplexDomainInstances listOfFlapLinkInstances;
    APMObjectDomainInstance selectedFlapLinkInstance;

    public MainFrame( )
    {

        super( "Flap Link Extensional Analysis" );
        setSize( 300 , 530 );


        // Create file menu
        bar = new MenuBar();
        fileMenu = new Menu( "File" );
        loadAPMDefinition = new MenuItem( "Load APM Definition" );
        loadData = new MenuItem( "Load Flap Link Data" );
        fileMenu.add( loadAPMDefinition );
        fileMenu.add( loadData );
        fileMenu.addSeparator( );
```

```
        exitProgram = new MenuItem( "Exit" );
        fileMenu.add( exitProgram );

        // Add listener to the menu items
        loadAPMDefinition.addActionListener( this );
        loadData.addActionListener( this );
        exitProgram.addActionListener( this );


        // Add menus to the bar
        bar.add( fileMenu );

        // Add bar
        setMenuBar( bar );



        setBackground( Color.lightGray );


        setVisible( true );


    }


    public void actionPerformed( ActionEvent e )
    {
        boolean success = false;
        FileDialog fileDialog;
        InfoDialog infoDialog;

        if( e.getSource() == loadAPMDefinition )
        {
            // Create file dialog and get the file name
            fileDialog = new FileDialog( this , "Select APM Definition File" , FileDialog.LOAD );
            fileDialog.show();
            lastDirectoryName = fileDialog.getDirectory();
            apmDefinitionFileName = lastDirectoryName + fileDialog.getFile();

            // Initialize the APMInterface
            APMInterface.initialize();

            success = APMInterface.loadAPMDefinitions( apmDefinitionFileName );

            // Display a dialog notifying wheter or not the APM definitions have been loaded
            if( success )
                infoDialog = new InfoDialog( this , "Message" , "APM loaded successfully" );
            else
                infoDialog = new InfoDialog( this , "Message" , "APM not loaded" );

            infoDialog.show();
        }


        else if( e.getSource() == loadData )
        {
            APMSourceSet sourceSetCursor;
            ListOfStrings listOfFileNames = new ListOfStrings();

            // Prompt user for a data file for each source set
            for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
            {
                sourceSetCursor = APMInterface.getSourceSets().elementAt( i );

                fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
            FileDialog.LOAD );
                fileDialog.setDirectory( lastDirectoryName );
```

```
            fileDialog.show();
            listOfFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
        }

        // Load the data
        success = APMInterface.loadSourceSetData( listOfFileNames );

        // Display a message indicating wheter or not the data was loaded succesfully
        if( success )
        {
            infoDialog = new InfoDialog( this , "Message" , "Data loaded successfully" );

            // Get the instances of flap_link
            listOfFlapLinkInstances = APMInterface.getInstancesOf( "flap_link" );

            // Set the default flap link instance to the first in the list of flap links
            selectedFlapLinkInstance = (APMObjectDomainInstance) listOfFlapLinkInstances.elementAt( 0 );

            // Create the top , infoDisplay and Bottom panels
            topPanel = new SelectionPanel( this );
            infoDisplayPanel = new InfoDisplayPanel( selectedFlapLinkInstance );
            bottomPanel = new Panel();


            // Set the layout manager for the bottom panel (a CardManager)
            cardManager = new CardLayout();
            bottomPanel.setLayout( cardManager );


            // Create and add bottom panel components (3 cards: canvas - formula based panel - fea based panel)
            // Card 1 :Empty canvas as the beginning bottom panel
            emptyCanvas = new Canvas();
            emptyCanvas.setBackground( Color.lightGray );
            bottomPanel.add( emptyCanvas , "Empty" );

            // Card 2: Formula Based Panel into the Bottom Panel
            formulaBasedAnalysisPanel = new FormulaBasedAnalysisPanel( selectedFlapLinkInstance );
            bottomPanel.add( formulaBasedAnalysisPanel , "Formula Based Panel" );

            // Card 3: FEA Based Panel into the Bottom Panel
            feaBasedAnalysisPanel = new FEABasedAnalysisPanel( selectedFlapLinkInstance );
            bottomPanel.add( feaBasedAnalysisPanel , "FEA Based Panel" );


            // Add the top , infoDisplay and bottom panels to the frame
            add( topPanel , BorderLayout.NORTH );
            add( infoDisplayPanel , BorderLayout.CENTER );
            add( bottomPanel , BorderLayout.SOUTH );

            setVisible( true );

        }
        else
            infoDialog = new InfoDialog( this , "Message" , "Data not loaded" );

        infoDialog.show();


    }

    else if( e.getSource() == exitProgram )
    {
        System.exit( 0 );
    }
  }

}
```

InfoDisplayPanel.java

```
import java.awt.*;
import apm.*;
import java.awt.event.*;

public class InfoDisplayPanel extends Panel
{
    private Label partNoLabel , lengthLabel , sleeve2CenterXLabel , materialNameLabel , ELabel , simpleALabel , detailedALabel , cteLabel;
    private TextField partNoField , lengthField , sleeve2CenterXField , materialNameField , EField , simpleAField , detailedAField , cteField;
    private GridBagLayout gbLayout;
    private GridBagConstraints gbConstraints;



    public InfoDisplayPanel( APMObjectDomainInstance flapLinkInstance )
    {
        gbLayout = new GridBagLayout();
        setLayout( gbLayout );

        gbConstraints = new GridBagConstraints();
        gbConstraints.anchor = GridBagConstraints.WEST;

        partNoLabel = new Label( "Part Number" );
        addComponent( partNoLabel , 0 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        partNoField = new TextField( 10 );
        partNoField.setEditable( false );
        partNoField.setBackground( Color.lightGray );
        addComponent( partNoField , 0 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        lengthLabel = new Label( "Effective Length" );
        addComponent( lengthLabel , 1 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        lengthField = new TextField( 10 );
        lengthField.setEditable( false );
        lengthField.setBackground( Color.lightGray );
        addComponent( lengthField , 1 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        sleeve2CenterXLabel = new Label( "Sleeve 2 Center.x" );
        addComponent( sleeve2CenterXLabel , 2 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        sleeve2CenterXField = new TextField( 10 );
        sleeve2CenterXField.setEditable( false );
        sleeve2CenterXField.setBackground( Color.lightGray );
        addComponent( sleeve2CenterXField , 2 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );


        materialNameLabel = new Label( "Material Name" );
        addComponent( materialNameLabel , 3 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        materialNameField = new TextField( 10 );
        materialNameField.setEditable( false );
        materialNameField.setBackground( Color.lightGray );
        addComponent( materialNameField, 3 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        ELabel = new Label( "E" );
        addComponent( ELabel , 4 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        EField = new TextField( 10 );
        EField.setEditable( false );
        EField.setBackground( Color.lightGray );
        addComponent( EField , 4 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        simpleALabel = new Label( "Simple Area" );
        addComponent( simpleALabel , 5 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        simpleAField = new TextField( 10 );
        simpleAField.setEditable( false );
        simpleAField.setBackground( Color.lightGray );
        addComponent( simpleAField , 5 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );
```

```
        detailedALabel = new Label( "Detailed Area" );
        addComponent( detailedALabel , 6 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        detailedAField = new TextField( 10 );
        detailedAField.setEditable( false );
        detailedAField.setBackground( Color.lightGray );
        addComponent( detailedAField , 6 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        cteLabel = new Label( "cte" );
        addComponent( cteLabel , 7 , 0 , 1 , 1 , 0 , 0 , 0 , 0 );
        cteField = new TextField( 10 );
        cteField.setBackground( Color.lightGray );
        cteField.setEditable( false );
        addComponent( cteField , 7 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        updateValues( flapLinkInstance );
        setVisible( true );
    }

    public void updateValues( APMObjectDomainInstance flapLinkInstance )
    {
        String partNumber = flapLinkInstance.
        getStringInstance( "part_number" ).
        getStringValue();

        double L = flapLinkInstance.
        getRealInstance( "effective_length" ).
        getRealValue();

        double x2 = flapLinkInstance.
        getObjectInstance( "sleeve_2" ).
        getObjectInstance( "center" ).
        getRealInstance( "x" ).
        getRealValue();

        String materialName =
        flapLinkInstance.
        getObjectInstance( "material" ).
        getStringInstance( "name" ).
        getStringValue();

        double E = flapLinkInstance.
        getObjectInstance( "material" ).
        getMultiLevelInstance( "stress_strain_model" ).
        getObjectInstance( "temperature_independent_linear_elastic" ).
        getRealInstance( "youngs_modulus" ).
        getRealValue( );

        double cte = flapLinkInstance.
        getObjectInstance( "material" ).
        getMultiLevelInstance( "stress_strain_model" ).
        getObjectInstance( "temperature_independent_linear_elastic" ).
        getRealInstance( "cte" ).
        getRealValue( );

        double simpleA = flapLinkInstance.
        getObjectInstance( "shaft" ).
        getMultiLevelInstance( "critical_cross_section" ).
        getObjectInstance( "simple" ).
        getRealInstance( "area" ).
        getRealValue( );

        double detailedA = flapLinkInstance.
        getObjectInstance( "shaft" ).
        getMultiLevelInstance( "critical_cross_section" ).
        getObjectInstance( "detailed" ).
        getRealInstance( "area" ).
        getRealValue( );
```

```
    partNoField.setText( partNumber );
    lengthField.setText( Double.toString( L ) );
    sleeve2CenterXField.setText( Double.toString( x2 ) );
    materialNameField.setText( materialName );
    EField.setText( Double.toString( E ) );
    simpleAField.setText( Double.toString( simpleA ) );
    detailedAField.setText( Double.toString( detailedA ) );
    cteField.setText( Double.toString( cte ) );

}

private void addComponent( Component c , int row , int column , int width , int height ,
int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
{
    gbConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
    // Set gridx and gridy
    gbConstraints.gridx = column;
    gbConstraints.gridy = row;

    // Set gridwidth and gridheight
    gbConstraints.gridwidth = width;
    gbConstraints.gridheight = height;

    // Set constraints
    gbLayout.setConstraints( c , gbConstraints );
    add( c );
}
```

SelectionPanel.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;

public class SelectionPanel extends Panel implements ActionListener , ItemListener
{
    private GridBagLayout gbLayout;
    private GridBagConstraints gbConstraints;
    private Label choiceButtonLabel;
    private Choice flapLinkInstancesList;
    private Button formulaBasedAnalysisButton;
    private Button FEABasedAnalysisButton;
    private MainFrame mainFrame;


    public SelectionPanel( MainFrame m )
    {
        mainFrame = m;


        // Set the layout manager for the panel (a GridLayoutBag)
        gbLayout = new GridBagLayout();
        gbConstraints = new GridBagConstraints();
        setLayout( gbLayout );

        // Label
        choiceButtonLabel = new Label( "Select Flap Link:" );
        addComponent( choiceButtonLabel , 0 , 0 , 1 , 1 , 20 , 0 , 10 , 0 );

        // Choice button (drop-down list) to select the flap link
        flapLinkInstancesList = new Choice();
        for( int i = 0 ; i < mainFrame.listOfFlapLinkInstances.size() ; i++ )
            flapLinkInstancesList.add(      mainFrame.listOfFlapLinkInstances.elementAt(      i      ).getStringInstance(      "part_number"
        ).getStringValue() );
```

```
        flapLinkInstancesList.addItemListener( this );
        gbConstraints.fill = GridBagConstraints.HORIZONTAL;
        addComponent( flapLinkInstancesList , 0 , 1 , 2 , 1 , 20 , 0 , 10 , 0 );

        // Formula-Based Button
        formulaBasedAnalysisButton = new Button( "Formula-Based" );
        addComponent( formulaBasedAnalysisButton , 1 , 0 , 1 , 1 , 0 , 5 , 0 , 5 );
        formulaBasedAnalysisButton.addActionListener( this );

        // FEA-Based Button
        FEABasedAnalysisButton = new Button( "FEA-Based" );
        addComponent( FEABasedAnalysisButton , 1 , 1 , 1 , 1 , 0 , 5 , 0 , 5 );
        FEABasedAnalysisButton.addActionListener( this );
    }

    public void actionPerformed( ActionEvent e )
    {
        if( e.getSource() == formulaBasedAnalysisButton )
            mainFrame.cardManager.show( mainFrame.bottomPanel , "Formula Based Panel" );

        if( e.getSource() == FEABasedAnalysisButton )
            mainFrame.cardManager.show( mainFrame.bottomPanel , "FEA Based Panel" );
    }


    public void itemStateChanged( ItemEvent e )
    {
        String selectedFlapLinkInstancePartNumber = flapLinkInstancesList.getSelectedItem();
        String aFlapLinkInstancePartNumber;
        APMObjectDomainInstance aFlapLinkInstance;


        for( int i = 0 ; i < mainFrame.listOfFlapLinkInstances.size() ; i++ )
        {
            aFlapLinkInstance = (APMObjectDomainInstance) mainFrame.listOfFlapLinkInstances.elementAt( i );
            aFlapLinkInstancePartNumber = aFlapLinkInstance.getStringInstance( "part_number" ).getStringValue();
            if( aFlapLinkInstancePartNumber.equals( selectedFlapLinkInstancePartNumber ) )
                mainFrame.selectedFlapLinkInstance = aFlapLinkInstance;
        }

        // Update the panels with the new flap link
        mainFrame.formulaBasedAnalysisPanel.updateValues( mainFrame.selectedFlapLinkInstance );
        mainFrame.feaBasedAnalysisPanel.updateValues( mainFrame.selectedFlapLinkInstance );
        mainFrame.infoDisplayPanel.updateValues( mainFrame.selectedFlapLinkInstance );



    }


    private void addComponent( Component component , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        gbConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        gbConstraints.gridx = column;
        gbConstraints.gridy = row;

        // Set gridwidth and gridheight
        gbConstraints.gridwidth = width;
        gbConstraints.gridheight = height;

        // Set constraints
        gbLayout.setConstraints( component , gbConstraints );
        add( component );
    }
```

FormulaBasedAnalysisPanel.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;



public class FormulaBasedAnalysisPanel extends java.awt.Panel implements java.awt.event.ActionListener, java.awt.event.ItemListener {
private Panel analVariablesPanel;
private Panel checkboxesPanel;
private Label forceLabel , deltaTLabel , deltaLLabel , stressLabel;
private TextField forceField , deltaTField , deltaLField , stressField ;

private GridBagLayout gbLayout;
private GridBagConstraints gbConstraints;
private double L , E , A , cte , deltaL , P , deltaT;

CheckboxGroup areaLevel;
Checkbox criticalSimple;
Checkbox criticalDetailed;
Button calculateButton;
APMObjectDomainInstance apmObjectDomainInstance;

private APMObjectDomainInstance flapLinkInstance;


public FormulaBasedAnalysisPanel( APMObjectDomainInstance apmObjectDomainInstance )
{
    flapLinkInstance = apmObjectDomainInstance;

    // GUI Creation:
    gbLayout = new GridBagLayout();
    setLayout( gbLayout );
    gbConstraints = new GridBagConstraints();


    // Creating a Panel for the Analysis Variables
    analVariablesPanel = new Panel();
    analVariablesPanel.setLayout( new GridLayout( 4 , 2 ) );

    forceLabel = new Label( "Force" );
    analVariablesPanel.add( forceLabel );
    forceField = new TextField( 10 );
    forceField.addActionListener( this );
    analVariablesPanel.add( forceField );

    deltaTLabel = new Label( "Delta T" );
    analVariablesPanel.add( deltaTLabel );
    deltaTField = new TextField( 10 );
    deltaTField.addActionListener( this );
    analVariablesPanel.add( deltaTField );

    deltaLLabel = new Label( "Delta L" );
    analVariablesPanel.add( deltaLLabel );
    deltaLField = new TextField( 10 );
    deltaLField.setEditable( false );
    analVariablesPanel.add( deltaLField );

    stressLabel = new Label( "Stress-X" );
    analVariablesPanel.add( stressLabel );
    stressField = new TextField( 10 );
    stressField.setEditable( false );
    analVariablesPanel.add( stressField );
```

```
    // Creating a Panel for the Checkbox group
    checkboxesPanel = new Panel();
    checkboxesPanel.setLayout( new GridLayout( 1 , 2 ) );

    // Create checkboxGroup to select the cross section level
    areaLevel = new CheckboxGroup();
    criticalDetailed = new Checkbox( "Critical-Detailed" , areaLevel , false );
    criticalDetailed.addItemListener( this );
    checkboxesPanel.add( criticalDetailed );
    criticalSimple = new Checkbox( "Critical-Simple" , areaLevel , true );
    criticalSimple.addItemListener( this );
    checkboxesPanel.add( criticalSimple );

    // Create a calculate button
    calculateButton = new Button( "Calculate" );
    calculateButton.addActionListener( this );


    gbConstraints.fill = GridBagConstraints.HORIZONTAL;

    addComponent( checkboxesPanel , 0 , 0 , 1 , 1 , 20 , 0 , 20 , 0 );
    addComponent( analVariablesPanel , 1 , 0 , 2 , 1 , 0 , 0 , 0 , 0);
    addComponent( calculateButton , 2 , 0 , 1 , 1 , 20 , 40 , 20 , 40);

    forceField.setText( "0" );
    deltaTField.setText( "0" );
    stressField.setText( "0" );
    updateValues( flapLinkInstance );
    setVisible( true );

}

public void updateValues( APMObjectDomainInstance flapLinkInstance )
{

    // Get the values of the attributes of the flap link
    L = flapLinkInstance.getRealInstance( "effective_length" ).getRealValue();

    E = flapLinkInstance.
    getObjectInstance( "material" ).
    getMultiLevelInstance( "stress_strain_model" ).
    getObjectInstance( "temperature_independent_linear_elastic" ).
    getRealInstance( "youngs_modulus" ).
    getRealValue( );

    cte = flapLinkInstance.
    getObjectInstance( "material" ).
    getMultiLevelInstance( "stress_strain_model" ).
    getObjectInstance( "temperature_independent_linear_elastic" ).
    getRealInstance( "cte" ).
    getRealValue( );

    A = flapLinkInstance.
    getObjectInstance( "shaft" ).
    getMultiLevelInstance( "critical_cross_section" ).
    getObjectInstance( "simple" ).
    getRealInstance( "area" ).
    getRealValue( );



    deltaLField.setText( "0" );
    stressField.setText( "0" );

}
```

```
    private void addComponent( Component c , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        gbConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        gbConstraints.gridx = column;
        gbConstraints.gridy = row;

        // Set gridwidth and gridheight
        gbConstraints.gridwidth = width;
        gbConstraints.gridheight = height;

        // Set constraints
        gbLayout.setConstraints( c , gbConstraints );
        add( c );
    }

    public void actionPerformed( ActionEvent e )
    {
        double P = Double.valueOf( forceField.getText() ).doubleValue();
        double deltaT = Double.valueOf( deltaTField.getText() ).doubleValue();
        double stress = Double.valueOf( stressField.getText() ).doubleValue();


        deltaL = ( P * L ) / ( E * A ) + cte * deltaT * L;

        stress = P/A;

        deltaLField.setText( Double.toString( deltaL ) );
        stressField.setText( Double.toString( stress ) );

    }

    public void itemStateChanged( ItemEvent e )
    {
        if( e.getSource() == criticalDetailed )
        {
            A = flapLinkInstance.
            getObjectInstance( "shaft" ).
            getMultiLevelInstance( "critical_cross_section" ).
            getObjectInstance( "detailed" ).
            getRealInstance( "area" ).
            getRealValue( );
        }

        if( e.getSource() == criticalSimple )
        {
            A = flapLinkInstance.
            getObjectInstance( "shaft" ).
            getMultiLevelInstance( "critical_cross_section" ).
            getObjectInstance( "simple" ).
            getRealInstance( "area" ).
            getRealValue( );
        }

        deltaLField.setText( "0" );
        stressField.setText( "0" );


    }

}
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import apm.*;
import file.*;
import gui.CloseWindow;

public class FEABasedAnalysisPanel extends java.awt.Panel implements ActionListener
{
    private TextArea outputArea;
    Hashtable keywordValuePairs;
    Button createAnsysFileButton;
    Button viewAnsysFileButton;
    Button runAnsysButton;
    Label forceLabel;
    TextField forceTextField;

    public FEABasedAnalysisPanel( APMObjectDomainInstance selectedFlapLink )
    {

        keywordValuePairs = new Hashtable( 15 );

        // Force label and field
        forceLabel = new Label( "Enter Force:" );
        add( forceLabel );
        forceTextField = new TextField( 10 );
        add( forceTextField );

        // Output area
        outputArea = new TextArea( "" , 5 , 30 , TextArea.SCROLLBARS_NONE );
        outputArea.setEditable( false );
        add( outputArea );

        // "Create Ansys file" button
        createAnsysFileButton = new Button( "Create ANSYS File" );
        add( createAnsysFileButton );
        createAnsysFileButton.addActionListener( this );

        // "View Ansys file" button
        viewAnsysFileButton = new Button( "View ANSYS File" );
        add( viewAnsysFileButton );
        viewAnsysFileButton.addActionListener( this );

        // "Run Ansys" button
        runAnsysButton = new Button( "Run ANSYS" );
        add( runAnsysButton );
        runAnsysButton.addActionListener( this );

        updateValues( selectedFlapLink );
    }

    public void updateValues( APMObjectDomainInstance selectedFlapLink )
    {
        keywordValuePairs = new Hashtable( 15 );

        // Get the values to be used in the ANSYS File and put the in an Hashtable

        String partNumber = selectedFlapLink.
        getStringInstance( "part_number" ).
        getStringValue();
        keywordValuePairs.put( "PARTNUMBER" , partNumber );

        double E = selectedFlapLink.
        getObjectInstance( "material" ).
        getMultiLevelInstance( "stress_strain_model" ).
        getObjectInstance( "temperature_independent_linear_elastic" ).
```

```
getRealInstance( "youngs_modulus" ).
getRealValue( );

keywordValuePairs.put( "EX" , String.valueOf( E ) );

double poissons = selectedFlapLink.
getObjectInstance( "material" ).
getMultiLevelInstance( "stress_strain_model" ).
getObjectInstance( "temperature_independent_linear_elastic" ).
getRealInstance( "poissons_ratio" ).
getRealValue( );
keywordValuePairs.put( "NIUX" , String.valueOf( poissons ) );

double L = selectedFlapLink.
getRealInstance( "effective_length" ).
getRealValue();
keywordValuePairs.put( "L" , String.valueOf( L ) );


double ws1 = selectedFlapLink.
getObjectInstance( "sleeve_1" ).
getRealInstance( "width" ).
getRealValue();
keywordValuePairs.put( "WS1" , String.valueOf( ws1 ) );

double ws2 = selectedFlapLink.
getObjectInstance( "sleeve_2" ).
getRealInstance( "width" ).
getRealValue();
keywordValuePairs.put( "WS2" , String.valueOf( ws2 ) );

double rs1 = selectedFlapLink.
getObjectInstance( "sleeve_1" ).
getRealInstance( "radius" ).
getRealValue();
keywordValuePairs.put( "RS1" , String.valueOf( rs1 ) );

double rs2 = selectedFlapLink.
getObjectInstance( "sleeve_2" ).
getRealInstance( "radius" ).
getRealValue();
keywordValuePairs.put( "RS2" , String.valueOf( rs2 ) );

double ts1 = selectedFlapLink.
getObjectInstance( "sleeve_1" ).
getRealInstance( "thickness" ).
getRealValue();
keywordValuePairs.put( "TS1" , String.valueOf( ts1 ) );

double ts2 = selectedFlapLink.
getObjectInstance( "sleeve_2" ).
getRealInstance( "thickness" ).
getRealValue();
keywordValuePairs.put( "TS2" , String.valueOf( ts2 ) );



double tw = selectedFlapLink.
getObjectInstance( "shaft" ).
getMultiLevelInstance( "critical_cross_section" ).
getObjectInstance( "simple" ).
getRealInstance( "tw" ).
getRealValue();
keywordValuePairs.put( "TW" , String.valueOf( tw ) );

double tf = selectedFlapLink.
getObjectInstance( "shaft" ).
getMultiLevelInstance( "critical_cross_section" ).
```

```
            getObjectInstance( "simple" ).
            getRealInstance( "tf" ).
            getRealValue();
            keywordValuePairs.put( "TF" , String.valueOf( tw ) );

            double wf = selectedFlapLink.
            getObjectInstance( "shaft" ).
            getMultiLevelInstance( "critical_cross_section" ).
            getObjectInstance( "simple" ).
            getRealInstance( "wf" ).
            getRealValue();
            keywordValuePairs.put( "WF" , String.valueOf( wf ) );

        }

        public void actionPerformed( ActionEvent e )
        {
            if( e.getSource() == createAnsysFileButton )
            {
                String force = forceTextField.getText();

                keywordValuePairs.put( "FORCE" , force );
                outputArea.appendText( "Creating ANSYS \"flaplink.dat\" ... \n" );
                FormFiller.fillForm( ".\\ansys\\form.dat" , ".\\ansys\\flaplink.dat" , '@' , keywordValuePairs );
                outputArea.appendText( "Done\n\n" );
            }

            else if( e.getSource() == viewAnsysFileButton )
            {
                FileViewer fileViewer = new FileViewer( ".\\ansys\\flaplink.dat" );
                fileViewer.addWindowListener( new CloseWindow() );
            }

            else if( e.getSource() == runAnsysButton )
            {

                // Launch Ansys
                Runtime runtimeObject = Runtime.getRuntime();
                try
                {
                    outputArea.appendText( "Starting Ansys..." );
                    runtimeObject.exec( "c:\\ansys\\bin\\ansysi.exe" );
                    System.exit( 0 );
                }
                catch ( IOException ioe )
                {
                    System.out.println( "Could not run Ansys..." );
                }
            }

        }

}
```

II.3 Back Plate Analysis and Synthesis Application

PlateExtensionalAnalysis.java

```
import gui.CloseWindowAndExit;
import apm.*;
import file.*;
import java.util.*;

public class PlateExtensionalAnalysis
{
    public static void main( String args[] )
    {
        MainFrame f = new MainFrame();
        f.addWindowListener( new CloseWindowAndExit() );
    }
}
```

MainFrame.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;
import java.util.*;
import java.io.*;
import file.*;
import gui.*;


/*
Frames and Panels in this application:

Container Type Layout Manager Contained by
--------- ---- -------------- ------------
MainFrame Frame GridBagLayout (none)
VariablesPanel Panel GridBagLayout MainFrame
AnalysisPanel Panel GridBagLayout MainFrame
RelationsPanel Panel GridBagLayout MainFrame

*/

public class MainFrame extends Frame implements ActionListener
{
    private GridBagConstraints mainFrameGridbagConstraints;
    private GridBagLayout mainFrameGridbagLayout;
    private MenuBar bar;
    private Menu fileMenu;
    private MenuItem loadAPMDefinition;
    private MenuItem loadData;
    private Menu saveDataSubMenu;
    private MenuItem saveLinkedPlateData;
    private MenuItem saveUnlinkedPlateData;
    private MenuItem exitProgram;
    private String apmDefinitionFileName;
    private String lastDirectoryName;
    ListOfAPMComplexDomainInstances listOfPlateInstances;
    APMObjectDomainInstance selectedPlateInstance;
    VariablesPanel variablesPanel;
    AnalysisPanel analysisPanel;
    RelationsPanel relationsPanel;
    InfoDialog infoDialog;

    public MainFrame( )
    {
        super( "Plate Extensional Analysis" );
```

```
    setSize( 760 , 600 );

    // Set the layout manager for the main frame (a GridLayoutBag)
    mainFrameGridbagLayout = new GridBagLayout();
    mainFrameGridbagConstraints = new GridBagConstraints();
    setLayout( mainFrameGridbagLayout );


    // Create file menu
    bar = new MenuBar();
    fileMenu = new Menu( "File" );
    loadAPMDefinition = new MenuItem( "Load APM Definition" );
    loadData = new MenuItem( "Load Plate Data" );
    saveDataSubMenu = new Menu( "Save Plate Data" );
    saveLinkedPlateData = new MenuItem( "Linked" );
    saveUnlinkedPlateData = new MenuItem( "Unlinked" );
    fileMenu.add( loadAPMDefinition );
    fileMenu.add( loadData );
    fileMenu.addSeparator();
    fileMenu.add( saveDataSubMenu );
    saveDataSubMenu.add( saveLinkedPlateData );
    saveDataSubMenu.add( saveUnlinkedPlateData );
    fileMenu.addSeparator( );
    exitProgram = new MenuItem( "Exit" );
    fileMenu.add( exitProgram );

    // Add listener to the menu items
    loadAPMDefinition.addActionListener( this );
    loadData.addActionListener( this );
    saveLinkedPlateData.addActionListener( this );
    saveUnlinkedPlateData.addActionListener( this );
    exitProgram.addActionListener( this );


    // Add menus to the bar
    bar.add( fileMenu );

    // Add bar
    setMenuBar( bar );


    setBackground( Color.lightGray );

    setVisible( true );


}


public void actionPerformed( ActionEvent e )
{
    boolean success = false;
    FileDialog fileDialog;

    if( e.getSource() == loadAPMDefinition )
    {
        // Create file dialog and get the file name
        fileDialog = new FileDialog( this , "Select APM Definition File" , FileDialog.LOAD );
        fileDialog.show();
        lastDirectoryName = fileDialog.getDirectory();
        apmDefinitionFileName = lastDirectoryName + fileDialog.getFile();

        // Initialize the APM Interface
        APMInterface.initialize();

        success = APMInterface.loadAPMDefinitions( apmDefinitionFileName );

        // Display a dialog notifying wheter or not the APM definitions have been loaded
```

```
    if( success )
    {
        // Create the panels that are going inside this main frame
        variablesPanel = new VariablesPanel( this );
        analysisPanel = new AnalysisPanel( this );
        relationsPanel = new RelationsPanel( this );

        // Add the panels to the frame
        addComponent( variablesPanel , 0 , 0 , 1 , 2 , 0 , 0 , 0 , 25 );
        addComponent( analysisPanel , 0 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );
        addComponent( relationsPanel , 1 , 1 , 1 , 1 , 0 , 0 , 0 , 0 );

        setVisible( true );

        infoDialog = new InfoDialog( this , "Message" , "APM loaded successfully" );
    }
    else
        infoDialog = new InfoDialog( this , "Message" , "APM not loaded" );

        infoDialog.show();
}


else if( e.getSource() == loadData )
{
    APMSourceSet sourceSetCursor;
    ListOfStrings listOfFileNames = new ListOfStrings();

    // Prompt user for a data file for each source set
    for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
    {
        sourceSetCursor = APMInterface.getSourceSets().elementAt( i );

        fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
    FileDialog.LOAD );
        fileDialog.setDirectory( lastDirectoryName );
        fileDialog.show();
        listOfFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
    }

    // Load the data
    success = APMInterface.loadSourceSetData( listOfFileNames );

    // Display a message indicating wheter or not the data was loaded succesfully
    if( success )
    {
        infoDialog = new InfoDialog( this , "Message" , "Data loaded successfully" );

        // Get the instances of plate
        listOfPlateInstances = APMInterface.getInstancesOf( "plate" );

        // Set the default plate instance to the first in the list of plates
        selectedPlateInstance = (APMObjectDomainInstance) listOfPlateInstances.elementAt( 0 );

        // Fill the instances choice menu
        for( int i = 0 ; i < listOfPlateInstances.size() ; i++ )
            variablesPanel.plateInstancesList.add( listOfPlateInstances.elementAt( i ).getStringInstance( "part_number"
        ).getStringValue() );


        // Display the data of the default plate
        variablesPanel.displayValues( selectedPlateInstance );

    }
    else
        infoDialog = new InfoDialog( this , "Message" , "Data not loaded" );

    infoDialog.show();
```

```
        }


        else if( e.getSource() == saveUnlinkedPlateData )
        {
            APMSourceSet sourceSetCursor;
            ListOfStrings listOfOutputFileNames = new ListOfStrings();

            // Prompt user for a data file for each source set
            for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
            {
                sourceSetCursor = APMInterface.getSourceSets().elementAt( i );
                fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
            FileDialog.LOAD );
                fileDialog.setDirectory( lastDirectoryName );
                fileDialog.show();
                listOfOutputFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
            }

            APMInterface.saveInstancesBySourceSet( listOfOutputFileNames );
        }

        else if( e.getSource() == saveLinkedPlateData )
        {
            fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
            fileDialog.setDirectory( lastDirectoryName );
            fileDialog.show();
            APMInterface.saveLinkedInstances( fileDialog.getDirectory() + fileDialog.getFile() );
        }



        else if( e.getSource() == exitProgram )
        {
            System.exit( 0 );
        }
    }



    private void addComponent( Component component , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        mainFrameGridbagConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        mainFrameGridbagConstraints.gridx = column;
        mainFrameGridbagConstraints.gridy = row;

        // Set gridwidth and gridheight
        mainFrameGridbagConstraints.gridwidth = width;
        mainFrameGridbagConstraints.gridheight = height;

        // Set constraints
        mainFrameGridbagLayout.setConstraints( component , mainFrameGridbagConstraints );
        add( component );
    }


}
```

VariablesPanel.java

```
import java.awt.*;
```

```
import java.awt.event.*;
import apm.*;
import gui.*;

public class VariablesPanel extends Panel implements ActionListener , ItemListener , TextListener
{
    private MainFrame mainFrame;
    private GridBagConstraints variablesPanelGridbagConstraints;
    private GridBagLayout variablesPanelGridbagLayout;
    Choice plateInstancesList; // Package access to make it accesible from main frame
    private Label inputLabel;
    private Label analysisInputLabel;
    private Label designerLabel;
    private Label materialLabel;
    private Label youngsModulusLabel;
    private Label L1Label;
    private Label L2Label;
    private Label L3Label;
    private Label lengthLabel;
    private Label widthLabel;
    private Label thicknessLabel;
    private Label hole1Label;
    private Label hole1DiameterLabel;
    private Label hole1CenterXLabel;
    private Label hole1CenterYLabel;
    private Label hole2Label;
    private Label hole2DiameterLabel;
    private Label hole2CenterXLabel;
    private Label hole2CenterYLabel;
    private Label criticalAreaLabel;
    private TextField designerField;
    private TextField materialField;
    TextField youngsModulusField; // Package access to make it accessible from analysisPanel
    private TextField L1Field;
    private TextField L2Field;
    private TextField L3Field;
    TextField lengthField; // Package access to make it accessible from analysisPanel
    private TextField widthField;
    private TextField thicknessField;
    private TextField hole1DiameterField;
    private TextField hole1CenterXField;
    private TextField hole1CenterYField;
    private TextField hole2DiameterField;
    private TextField hole2CenterXField;
    private TextField hole2CenterYField;
    TextField criticalAreaField; // Package access to make it accessible from analysisPanel
    private Checkbox designerIsInputCheckbox;
    private Checkbox materialIsInputCheckbox;
    private Checkbox youngsModulusIsInputCheckbox;
    Checkbox youngsModulusIsAnalysisInputCheckbox; // Package access to make it accessible from analysisPanel
    private Checkbox L1IsInputCheckbox;
    private Checkbox L2IsInputCheckbox;
    private Checkbox L3IsInputCheckbox;
    private Checkbox lengthIsInputCheckbox;
    Checkbox lengthIsAnalysisInputCheckbox; // Package access to make it accessible from analysisPanel
    private Checkbox widthIsInputCheckbox;
    private Checkbox thicknessIsInputCheckbox;
    private Checkbox hole1DiameterIsInputCheckbox;
    private Checkbox hole1CenterXIsInputCheckbox;
    private Checkbox hole1CenterYIsInputCheckbox;
    private Checkbox hole2DiameterIsInputCheckbox;
    private Checkbox hole2CenterXIsInputCheckbox;
    private Checkbox hole2CenterYIsInputCheckbox;
    private Checkbox criticalAreaIsInputCheckbox;
    Checkbox criticalAreaIsAnalysisInputCheckbox; // Package access to make it accessible from analysisPanel

    private Button solveAPMButton;
```

```
private APMStringInstance designerInstance;
private APMStringInstance materialInstance;
private APMRealInstance youngsModulusInstance;
private APMRealInstance L1Instance;
private APMRealInstance L2Instance;
private APMRealInstance L3Instance;
private APMRealInstance lengthInstance;
private APMRealInstance widthInstance;
private APMRealInstance thicknessInstance;
private APMRealInstance d1Instance;
private APMRealInstance x1Instance;
private APMRealInstance y1Instance;
private APMRealInstance d2Instance;
private APMRealInstance x2Instance;
private APMRealInstance y2Instance;
private APMRealInstance criticalAreaInstance;


public VariablesPanel( MainFrame m )
{
    mainFrame = m;

    // Set the layout manager for the panel (a GridLayoutBag)
    variablesPanelGridbagLayout = new GridBagLayout();
    variablesPanelGridbagConstraints = new GridBagConstraints();
    setLayout( variablesPanelGridbagLayout );

    // Start a row counter for component placement
    int row = 0;

    // Drop down list of plates
    Label choiceButtonLabel = new Label( "Select Plate:" );
    choiceButtonLabel.setAlignment( Label.RIGHT );
    addComponent( choiceButtonLabel , row , 0 , 1 , 1 , 10 , 0 , 10 , 0 );
    plateInstancesList = new Choice();
    plateInstancesList.addItemListener( this );
    variablesPanelGridbagConstraints.fill = GridBagConstraints.HORIZONTAL; // Do not grow taller when resizing window
    addComponent( plateInstancesList , row , 1 , 2 , 1 , 0 , 10 , 0 , 0 );


    // Label for the Input checkboxes
    inputLabel = new Label( "Input" );
    inputLabel.setAlignment( Label.CENTER );
    addComponent( inputLabel , ++row , 2 , 1 , 1 , 0 , 0 , 0 , 0 );

    // Label for the Analysis Input checkboxes
    analysisInputLabel = new Label( "AI" );
    analysisInputLabel.setAlignment( Label.CENTER );
    addComponent( analysisInputLabel , row , 3 , 1 , 1 , 0 , 0 , 0 , 0 );

    // Designer row
    designerLabel = new Label( "Designer" );
    designerLabel.setAlignment( Label.RIGHT );
    addComponent( designerLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    designerField = new TextField( 20 );
    designerField.setBackground( Color.lightGray );
    addComponent( designerField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
    designerIsInputCheckbox = new Checkbox( );
    addComponent( designerIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

    // Material row
    materialLabel = new Label( "Material" );
    materialLabel.setAlignment( Label.RIGHT );
    addComponent( materialLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    materialField = new TextField( 10 );
    materialField.setBackground( Color.lightGray );
    addComponent( materialField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
    materialIsInputCheckbox = new Checkbox( );
```

```
addComponent( materialIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// Youngs Modulus row
youngsModulusLabel = new Label( "Youngs Modulus" );
youngsModulusLabel.setAlignment( Label.RIGHT );
addComponent( youngsModulusLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
youngsModulusField = new TextField( 10 );
youngsModulusField.setBackground( Color.lightGray );
addComponent( youngsModulusField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
youngsModulusIsInputCheckbox = new Checkbox( );
addComponent( youngsModulusIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );
youngsModulusIsAnalysisInputCheckbox = new Checkbox( );
addComponent( youngsModulusIsAnalysisInputCheckbox , row , 3 , 1 , 1 , 0 , 5 , 0 , 0 );


// L1 row
L1Label = new Label( "L1" );
L1Label.setAlignment( Label.RIGHT );
addComponent( L1Label , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
L1Field = new TextField( 10 );
L1Field.setBackground( Color.lightGray );
addComponent( L1Field , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
L1IsInputCheckbox = new Checkbox( );
addComponent( L1IsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// L2 row
L2Label = new Label( "L2" );
L2Label.setAlignment( Label.RIGHT );
addComponent( L2Label , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
L2Field = new TextField( 10 );
L2Field.setBackground( Color.lightGray );
addComponent( L2Field , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
L2IsInputCheckbox = new Checkbox( );
addComponent( L2IsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// L3 row
L3Label = new Label( "L3" );
L3Label.setAlignment( Label.RIGHT );
addComponent( L3Label , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
L3Field = new TextField( 10 );
L3Field.setBackground( Color.lightGray );
addComponent( L3Field , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
L3IsInputCheckbox = new Checkbox( );
addComponent( L3IsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );


// Length row
lengthLabel = new Label( "Length" );
lengthLabel.setAlignment( Label.RIGHT );
addComponent( lengthLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
lengthField = new TextField( 10 );
lengthField.setBackground( Color.lightGray );
addComponent( lengthField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
lengthIsInputCheckbox = new Checkbox( );
addComponent( lengthIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );
lengthIsAnalysisInputCheckbox = new Checkbox( );
addComponent( lengthIsAnalysisInputCheckbox , row , 3 , 1 , 1 , 0 , 5 , 0 , 0 );


// Width row
widthLabel = new Label( "Width" );
widthLabel.setAlignment( Label.RIGHT );
addComponent( widthLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
widthField = new TextField( 10 );
widthField.setBackground( Color.lightGray );
addComponent( widthField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
widthIsInputCheckbox = new Checkbox( );
addComponent( widthIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );
```

```
// Thickness row
thicknessLabel = new Label( "Thickness" );
thicknessLabel.setAlignment( Label.RIGHT );
addComponent( thicknessLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
thicknessField = new TextField( 10 );
thicknessField.setBackground( Color.lightGray );
addComponent( thicknessField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
thicknessIsInputCheckbox = new Checkbox( );
addComponent( thicknessIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// Critical area row
criticalAreaLabel = new Label( "Critical Area" );
criticalAreaLabel.setAlignment( Label.RIGHT );
addComponent( criticalAreaLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
criticalAreaField = new TextField( 10 );
criticalAreaField.setBackground( Color.lightGray );
addComponent( criticalAreaField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
criticalAreaIsInputCheckbox = new Checkbox( );
addComponent( criticalAreaIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );
criticalAreaIsAnalysisInputCheckbox = new Checkbox( );
addComponent( criticalAreaIsAnalysisInputCheckbox , row , 3 , 1 , 1 , 0 , 5 , 0 , 0 );

// Hole 1 label
hole1Label = new Label( "Hole 1" );
hole1Label.setAlignment( Label.CENTER );
addComponent( hole1Label , ++row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );

// Hole 1 diameter row
hole1DiameterLabel = new Label( "Diameter" );
hole1DiameterLabel.setAlignment( Label.RIGHT );
addComponent( hole1DiameterLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
hole1DiameterField = new TextField( 10 );
hole1DiameterField.setBackground( Color.lightGray );
addComponent( hole1DiameterField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
hole1DiameterIsInputCheckbox = new Checkbox( );
addComponent( hole1DiameterIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// Hole 1 center x row
hole1CenterXLabel = new Label( "Center X" );
hole1CenterXLabel.setAlignment( Label.RIGHT );
addComponent( hole1CenterXLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
hole1CenterXField = new TextField( 10 );
hole1CenterXField.setBackground( Color.lightGray );
addComponent( hole1CenterXField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
hole1CenterXIsInputCheckbox = new Checkbox( );
addComponent( hole1CenterXIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// Hole 1 center y row
hole1CenterYLabel = new Label( "Center Y" );
hole1CenterYLabel.setAlignment( Label.RIGHT );
addComponent( hole1CenterYLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
hole1CenterYField = new TextField( 10 );
hole1CenterYField.setBackground( Color.lightGray );
addComponent( hole1CenterYField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
hole1CenterYIsInputCheckbox = new Checkbox( );
addComponent( hole1CenterYIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

// Hole 2 label
hole2Label = new Label( "Hole 2" );
hole2Label.setAlignment( Label.CENTER );
addComponent( hole2Label , ++row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );

// Hole 2 diameter row
hole2DiameterLabel = new Label( "Diameter" );
hole2DiameterLabel.setAlignment( Label.RIGHT );
addComponent( hole2DiameterLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
```

```
    hole2DiameterField = new TextField( 10 );
    hole2DiameterField.setBackground( Color.lightGray );
    addComponent( hole2DiameterField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
    hole2DiameterIsInputCheckbox = new Checkbox( );
    addComponent( hole2DiameterIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

    // Hole 2 center x row
    hole2CenterXLabel = new Label( "Center X" );
    hole2CenterXLabel.setAlignment( Label.RIGHT );
    addComponent( hole2CenterXLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    hole2CenterXField = new TextField( 10 );
    hole2CenterXField.setBackground( Color.lightGray );
    addComponent( hole2CenterXField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
    hole2CenterXIsInputCheckbox = new Checkbox( );
    addComponent( hole2CenterXIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

    // Hole 2 center y row
    hole2CenterYLabel = new Label( "Center Y" );
    hole2CenterYLabel.setAlignment( Label.RIGHT );
    addComponent( hole2CenterYLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    hole2CenterYField = new TextField( 10 );
    hole2CenterYField.setBackground( Color.lightGray );
    addComponent( hole2CenterYField , row , 1 , 1 , 1 , 0 , 5 , 0 , 0 );
    hole2CenterYIsInputCheckbox = new Checkbox( );
    addComponent( hole2CenterYIsInputCheckbox , row , 2 , 1 , 1 , 0 , 5 , 0 , 0 );

    // SolveAPM button
    solveAPMButton = new Button( "Solve APM" );
    addComponent( solveAPMButton , ++row , 1 , 1 , 1 , 0 , 10 , 20 , 0 );


    // Set attribute name labels background colors according to the attribute type
    colorAttributeNameLabels();


    // Add item listeners to check boxes
    designerIsInputCheckbox.addItemListener( this );
    materialIsInputCheckbox.addItemListener( this );
    youngsModulusIsInputCheckbox.addItemListener( this );
    L1IsInputCheckbox.addItemListener( this );
    L2IsInputCheckbox.addItemListener( this );
    L3IsInputCheckbox.addItemListener( this );
    lengthIsInputCheckbox.addItemListener( this );
    widthIsInputCheckbox.addItemListener( this );
    thicknessIsInputCheckbox.addItemListener( this );
    hole1DiameterIsInputCheckbox.addItemListener( this );
    hole1CenterXIsInputCheckbox.addItemListener( this );
    hole1CenterYIsInputCheckbox.addItemListener( this );
    hole2DiameterIsInputCheckbox.addItemListener( this );
    hole2CenterXIsInputCheckbox.addItemListener( this );
    hole2CenterYIsInputCheckbox.addItemListener( this );
    criticalAreaIsInputCheckbox.addItemListener( this );


    // Add action listener to solve button
    solveAPMButton.addActionListener( this );



}



public void actionPerformed( ActionEvent e )
{
    // SolveAPM button has been pressed: Solve for variables
    if( e.getSource() == solveAPMButton )
```

```
    {
        solveValues( mainFrame.selectedPlateInstance );
        mainFrame.infoDialog = new InfoDialog( mainFrame , "Message" , "Done" );
        mainFrame.infoDialog.show();
    }
}

public void textValueChanged( TextEvent e )
{
    // A value field has changed. Assign the new value to the corresponding instance
    if( ( e.getSource() == designerField ) && !designerField.getText().equals( "" ) )
        designerInstance.setValue( designerField.getText() );
    else if( ( e.getSource() == materialField ) && !materialField.getText().equals( "" ) )
        materialInstance.setValue( materialField.getText() );
    else if( ( e.getSource() == youngsModulusField ) && !youngsModulusField.getText().equals( "" ) )
        youngsModulusInstance.setValue( Double.valueOf( youngsModulusField.getText() ).doubleValue() );
    else if( ( e.getSource() == L1Field ) && !L1Field.getText().equals( "" ) )
        L1Instance.setValue( Double.valueOf( L1Field.getText() ).doubleValue() );
    else if( ( e.getSource() == L2Field ) && !L2Field.getText().equals( "" ) )
        L2Instance.setValue( Double.valueOf( L2Field.getText() ).doubleValue() );
    else if( ( e.getSource() == L3Field ) && !L3Field.getText().equals( "" ) )
        L3Instance.setValue( Double.valueOf( L3Field.getText() ).doubleValue() );
    else if( ( e.getSource() == lengthField ) && !lengthField.getText().equals( "" ) )
        lengthInstance.setValue( Double.valueOf( lengthField.getText() ).doubleValue() );
    else if( ( e.getSource() == widthField ) && !widthField.getText().equals( "" ) )
        widthInstance.setValue( Double.valueOf( widthField.getText() ).doubleValue() );
    else if( ( e.getSource() == thicknessField ) && !thicknessField.getText().equals( "" ) )
        thicknessInstance.setValue( Double.valueOf( thicknessField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole1DiameterField ) && !hole1DiameterField.getText().equals( "" ) )
        d1Instance.setValue( Double.valueOf( hole1DiameterField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole1CenterXField ) && !hole1CenterXField.getText().equals( "" ) )
        x1Instance.setValue( Double.valueOf( hole1CenterXField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole1CenterYField ) && !hole1CenterYField.getText().equals( "" ) )
        y1Instance.setValue( Double.valueOf( hole1CenterYField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole2DiameterField ) && !hole2DiameterField.getText().equals( "" ) )
        d2Instance.setValue( Double.valueOf( hole2DiameterField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole2CenterXField ) && !hole2CenterXField.getText().equals( "" ) )
        x2Instance.setValue( Double.valueOf( hole2CenterXField.getText() ).doubleValue() );
    else if( ( e.getSource() == hole2CenterYField ) && !hole2CenterYField.getText().equals( "" ) )
        y2Instance.setValue( Double.valueOf( hole2CenterYField.getText() ).doubleValue() );
    else if( ( e.getSource() == criticalAreaField ) && !criticalAreaField.getText().equals( "" ) )
        criticalAreaInstance.setValue( Double.valueOf( criticalAreaField.getText() ).doubleValue() );

}



public void itemStateChanged( ItemEvent e )
{
    String selectedPlateInstancePartNumber = plateInstancesList.getSelectedItem();
    String aPlateInstancePartNumber;
    APMObjectDomainInstance aPlateInstance;

    // The selected plate instance has changed
    if( e.getSource() == plateInstancesList )
    {
        for( int i = 0 ; i < mainFrame.listOfPlateInstances.size() ; i++ )
        {
            aPlateInstance = (APMObjectDomainInstance) mainFrame.listOfPlateInstances.elementAt( i );
            aPlateInstancePartNumber = aPlateInstance.getStringInstance( "part_number" ).getStringValue();
            if( aPlateInstancePartNumber.equals( selectedPlateInstancePartNumber ) )
                mainFrame.selectedPlateInstance = aPlateInstance;
        }

        // Display the available values
        displayValues( mainFrame.selectedPlateInstance );
    }
```

```
// An isInput checkbox has been clicked.
else if( e.getSource() == designerIsInputCheckbox )
{
    if( designerIsInputCheckbox.getState() == true )
        designerInstance.setAsInput();
    else
        designerInstance.setAsOutput();

    designerField.setEditable( designerIsInputCheckbox.getState() );
}
else if( e.getSource() == materialIsInputCheckbox )
{
    if( materialIsInputCheckbox.getState() == true )
        materialInstance.setAsInput();
else
    materialInstance.setAsOutput();

materialField.setEditable( materialIsInputCheckbox.getState() );
}
else if( e.getSource() == youngsModulusIsInputCheckbox )
{
    if( youngsModulusIsInputCheckbox.getState() == true )
    youngsModulusInstance.setAsInput();
else
    youngsModulusInstance.setAsOutput();

    youngsModulusField.setEditable( youngsModulusIsInputCheckbox.getState() );
}

else if( e.getSource() == L1IsInputCheckbox )
{
    if( L1IsInputCheckbox.getState() == true )
        L1Instance.setAsInput();
    else
        L1Instance.setAsOutput();

    L1Field.setEditable( L1IsInputCheckbox.getState() );
}
else if( e.getSource() == L2IsInputCheckbox )
{
    if( L2IsInputCheckbox.getState() == true )
        L2Instance.setAsInput();
    else
        L2Instance.setAsOutput();

    L2Field.setEditable( L2IsInputCheckbox.getState() );
}
else if( e.getSource() == L3IsInputCheckbox )
{
    if( L3IsInputCheckbox.getState() == true )
        L3Instance.setAsInput();
    else
        L3Instance.setAsOutput();

    L3Field.setEditable( L3IsInputCheckbox.getState() );
}
else if( e.getSource() == lengthIsInputCheckbox )
{
    if( lengthIsInputCheckbox.getState() == true )
        lengthInstance.setAsInput();
    else
        lengthInstance.setAsOutput();

    lengthField.setEditable( lengthIsInputCheckbox.getState() );
}
else if( e.getSource() == widthIsInputCheckbox )
{
    if( widthIsInputCheckbox.getState() == true )
```

```
            widthInstance.setAsInput();
        else
            widthInstance.setAsOutput();

        widthField.setEditable( widthIsInputCheckbox.getState() );
    }
    else if( e.getSource() == thicknessIsInputCheckbox )
    {
        if( thicknessIsInputCheckbox.getState() == true )
            thicknessInstance.setAsInput();
        else
            thicknessInstance.setAsOutput();

        thicknessField.setEditable( thicknessIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole1DiameterIsInputCheckbox )
    {
        if( hole1DiameterIsInputCheckbox.getState() == true )
            d1Instance.setAsInput();
        else
            d1Instance.setAsOutput();

        hole1DiameterField.setEditable( hole1DiameterIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole1CenterXIsInputCheckbox )
    {
        if( hole1CenterXIsInputCheckbox.getState() == true )
            x1Instance.setAsInput();
        else
            x1Instance.setAsOutput();

        hole1CenterXField.setEditable( hole1CenterXIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole1CenterYIsInputCheckbox )
    {
        if( hole1CenterYIsInputCheckbox.getState() == true )
            y1Instance.setAsInput();
        else
            y1Instance.setAsOutput();

        hole1CenterYField.setEditable( hole1CenterYIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole2DiameterIsInputCheckbox )
    {
        if( hole2DiameterIsInputCheckbox.getState() == true )
            d2Instance.setAsInput();
        else
            d2Instance.setAsOutput();

        hole2DiameterField.setEditable( hole2DiameterIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole2CenterXIsInputCheckbox )
    {
        if( hole2CenterXIsInputCheckbox.getState() == true )
            x2Instance.setAsInput();
        else
            x2Instance.setAsOutput();

        hole2CenterXField.setEditable( hole2CenterXIsInputCheckbox.getState() );
    }
    else if( e.getSource() == hole2CenterYIsInputCheckbox )
    {
        if( hole2CenterYIsInputCheckbox.getState() == true )
            y2Instance.setAsInput();
        else
            y2Instance.setAsOutput();

        hole2CenterYField.setEditable( hole2CenterYIsInputCheckbox.getState() );
```

```
    }
    else if( e.getSource() == criticalAreaIsInputCheckbox )
    {
        if( criticalAreaIsInputCheckbox.getState() == true )
            criticalAreaInstance.setAsInput();
        else
            criticalAreaInstance.setAsOutput();

        criticalAreaField.setEditable( criticalAreaIsInputCheckbox.getState() );
    }



}


public void displayValues( APMObjectDomainInstance plateInstance )
{

    // Clear fields contents
    designerField.setText( "" );
    materialField.setText( "" );
    youngsModulusField.setText( "" );
    L1Field.setText( "" );
    L2Field.setText( "" );
    L3Field.setText( "" );
    lengthField.setText( "" );
    widthField.setText( "" );
    thicknessField.setText( "" );
    hole1DiameterField.setText( "" );
    hole1CenterXField.setText( "" );
    hole1CenterYField.setText( "" );
    hole2DiameterField.setText( "" );
    hole2CenterXField.setText( "" );
    hole2CenterYField.setText( "" );
    criticalAreaField.setText( "" );


    // Designer Field
    designerInstance = plateInstance.
    getObjectInstance( "designer" ).
    getStringInstance( "first_name" );

    if( designerInstance.hasValue() )
    {
        String designerName = designerInstance.getStringValue() + " " +
        plateInstance.getObjectInstance( "designer" ).getStringInstance( "last_name" ).getStringValue();
        designerField.setText( designerName );
    }


    // Material Field
    materialInstance = plateInstance.
    getObjectInstance( "material" ).
    getStringInstance( "materialName" );

    if( materialInstance.hasValue() )
    {
        String materialName = materialInstance.getStringValue();
        materialField.setText( materialName );
    }

    // Youngs Modulus Field
    youngsModulusInstance = plateInstance.
    getObjectInstance( "material" ).
    getRealInstance( "youngsModulus" );

    if( youngsModulusInstance.hasValue() )
```

```
{
    double youngsModulus = youngsModulusInstance.getRealValue();
    youngsModulusField.setText( Double.toString( youngsModulus ) );
}

// L1 Field
L1Instance = plateInstance.
getRealInstance( "l1" );

if( L1Instance.hasValue() )
{
    double L1 = L1Instance.getRealValue();
    L1Field.setText( Double.toString( L1 ) );
}


// L2 Field
L2Instance = plateInstance.
getRealInstance( "l2" );

if( L2Instance.hasValue() )
{
    double L2 = L2Instance.getRealValue();
    L2Field.setText( Double.toString( L2 ) );
}


// L3 Field
L3Instance = plateInstance.
getRealInstance( "l3" );

if( L3Instance.hasValue() )
{
    double L3 = L3Instance.getRealValue();
    L3Field.setText( Double.toString( L3 ) );
}


// Length Field
lengthInstance = plateInstance.
getRealInstance( "length" );

if( lengthInstance.hasValue() )
{
    double length = lengthInstance.getRealValue();
    lengthField.setText( Double.toString( length ) );
}


// Width Field
widthInstance = plateInstance.
getRealInstance( "width" );

if( widthInstance.hasValue() )
{
    double width = widthInstance.getRealValue();
    widthField.setText( Double.toString( width ) );
}


// Thickness Field
thicknessInstance = plateInstance.
getRealInstance( "thickness" );

if( thicknessInstance.hasValue() )
{
    double thickness = thicknessInstance.getRealValue();
    thicknessField.setText( Double.toString( thickness ) );
```

```
}

// Hole 1 Diameter Field
d1Instance = plateInstance.
getObjectInstance( "hole1" ).
getRealInstance( "diameter" );

if( d1Instance.hasValue() )
{
    double d1 = d1Instance.getRealValue();
    hole1DiameterField.setText( Double.toString( d1 ) );
}

// Hole 1 Center X Field
x1Instance = plateInstance.
getObjectInstance( "hole1" ).
getObjectInstance( "center" ).
getRealInstance( "x" );

if( x1Instance.hasValue() )
{
    double x1 = x1Instance.getRealValue();
    hole1CenterXField.setText( Double.toString( x1 ) );
}

// Hole 1 Center Y Field
y1Instance = plateInstance.
getObjectInstance( "hole1" ).
getObjectInstance( "center" ).
getRealInstance( "y" );

if( y1Instance.hasValue() )
{
    double y1 = y1Instance.getRealValue();
    hole1CenterYField.setText( Double.toString( y1 ) );
}

// Hole 2 Diameter Field
d2Instance = plateInstance.
getObjectInstance( "hole2" ).
getRealInstance( "diameter" );

if( d2Instance.hasValue() )
{
    double d2 = d2Instance.getRealValue();
    hole2DiameterField.setText( Double.toString( d2 ) );
}

// Hole 2 Center X Field
x2Instance = plateInstance.
getObjectInstance( "hole2" ).
getObjectInstance( "center" ).
getRealInstance( "x" );

if( x2Instance.hasValue() )
{
    double x2 = x2Instance.getRealValue();
    hole2CenterXField.setText( Double.toString( x2 ) );
}

// Hole 2 Center Y Field
y2Instance = plateInstance.
```

```
getObjectInstance( "hole2" ).
getObjectInstance( "center" ).
getRealInstance( "y" );

if( y2Instance.hasValue() )
{
    double y2 = y2Instance.getRealValue();
    hole2CenterYField.setText( Double.toString( y2 ) );
}


// Critical Area Field
criticalAreaInstance = plateInstance.
getRealInstance( "critical_area" );

if( criticalAreaInstance.hasValue() )
{
    double criticalArea = criticalAreaInstance.getRealValue();
    criticalAreaField.setText( Double.toString( criticalArea ) );
}



// Set the isInput checkboxes for each variable
designerIsInputCheckbox.setState( designerInstance.isInput() );
materialIsInputCheckbox.setState( materialInstance.isInput() );
youngsModulusIsInputCheckbox.setState( youngsModulusInstance.isInput() );
L1IsInputCheckbox.setState( L1Instance.isInput() );
L2IsInputCheckbox.setState( L2Instance.isInput() );
L3IsInputCheckbox.setState( L3Instance.isInput() );
lengthIsInputCheckbox.setState( lengthInstance.isInput() );
widthIsInputCheckbox.setState( widthInstance.isInput() );
thicknessIsInputCheckbox.setState( thicknessInstance.isInput() );
hole1DiameterIsInputCheckbox.setState( d1Instance.isInput() );
hole1CenterXIsInputCheckbox.setState( x1Instance.isInput() );
hole1CenterYIsInputCheckbox.setState( y1Instance.isInput() );
hole2DiameterIsInputCheckbox.setState( d2Instance.isInput() );
hole2CenterXIsInputCheckbox.setState( x2Instance.isInput() );
hole2CenterYIsInputCheckbox.setState( y2Instance.isInput() );
criticalAreaIsInputCheckbox.setState( criticalAreaInstance.isInput() );

// Set the set editable property for each variable field
designerField.setEditable( designerInstance.isInput() );
materialField.setEditable( materialInstance.isInput() );
youngsModulusField.setEditable( youngsModulusInstance.isInput() );
L1Field.setEditable( L1Instance.isInput() );
L2Field.setEditable( L2Instance.isInput() );
L3Field.setEditable( L3Instance.isInput() );
lengthField.setEditable( lengthInstance.isInput() );
widthField.setEditable( widthInstance.isInput() );
thicknessField.setEditable( thicknessInstance.isInput() );
hole1DiameterField.setEditable( d1Instance.isInput() );
hole1CenterXField.setEditable( x1Instance.isInput() );
hole1CenterYField.setEditable( y1Instance.isInput() );
hole2DiameterField.setEditable( d2Instance.isInput() );
hole2CenterXField.setEditable( x2Instance.isInput() );
hole2CenterYField.setEditable( y2Instance.isInput() );
criticalAreaField.setEditable( criticalAreaInstance.isInput() );

// Add text listeners to value fields to detect a change in value
designerField.addTextListener( this );
materialField.addTextListener( this );
youngsModulusField.addTextListener( this );
L1Field.addTextListener( this );
L2Field.addTextListener( this );
L3Field.addTextListener( this );
lengthField.addTextListener( this );
widthField.addTextListener( this );
```

```
    thicknessField.addTextListener( this );
    hole1DiameterField.addTextListener( this );
    hole1CenterXField.addTextListener( this );
    hole1CenterYField.addTextListener( this );
    hole2DiameterField.addTextListener( this );
    hole2CenterXField.addTextListener( this );
    hole2CenterYField.addTextListener( this );
    criticalAreaField.addTextListener( this );


}

private void solveValues( APMObjectDomainInstance plateInstance )
{
    // Youngs Modulus value
    if( youngsModulusInstance.isOutput() )
    {
        APMRealInstance youngsModulusInstance = plateInstance.
        getObjectInstance( "material" ).
        getRealInstance( "youngsModulus" );
        setFieldValue( youngsModulusInstance , youngsModulusField );
    }


    // L1 value
    if( L1Instance.isOutput() )
    {
        APMRealInstance L1Instance = plateInstance.
        getRealInstance( "l1" );
        setFieldValue( L1Instance , L1Field );
    }



    // L2 value
    if( L2Instance.isOutput() )
    {
        APMRealInstance L2Instance = plateInstance.
        getRealInstance( "l2" );
        setFieldValue( L2Instance , L2Field );
    }

    // L3 value
    if( L3Instance.isOutput() )
    {
        APMRealInstance L3Instance = plateInstance.
        getRealInstance( "l3" );
        setFieldValue( L3Instance , L3Field );
    }


    // Length value
    if( lengthInstance.isOutput() )
    {
        APMRealInstance lengthInstance = plateInstance.
        getRealInstance( "length" );
        setFieldValue( lengthInstance , lengthField );
    }


    // Width value
    if( widthInstance.isOutput() )
    {
        APMRealInstance widthInstance = plateInstance.
        getRealInstance( "width" );
        setFieldValue( widthInstance , widthField );
    }
```

```
// Thickness value
if( thicknessInstance.isOutput() )
{
    APMRealInstance thicknessInstance = plateInstance.
    getRealInstance( "thickness" );
    setFieldValue( thicknessInstance , thicknessField );
}

// Critical Area Value
if( criticalAreaInstance.isOutput() )
{
    APMRealInstance criticalAreaInstance = plateInstance.
    getRealInstance( "critical_area" );
    setFieldValue( criticalAreaInstance , criticalAreaField );
}


// Hole 1 diameter value
if( d1Instance.isOutput() )
{
    APMRealInstance d1Instance = plateInstance.
    getObjectInstance( "hole1" ).
    getRealInstance( "diameter" );
    setFieldValue( d1Instance , hole1DiameterField );
}

// Hole 1 center X value
if( x1Instance.isOutput() )
{
    APMRealInstance x1Instance = plateInstance.
    getObjectInstance( "hole1" ).
    getObjectInstance( "center" ).
    getRealInstance( "x" );
    setFieldValue( x1Instance , hole1CenterXField );
}

// Hole 1 center Y value
if( y1Instance.isOutput() )
{
    APMRealInstance y1Instance = plateInstance.
    getObjectInstance( "hole1" ).
    getObjectInstance( "center" ).
    getRealInstance( "y" );
    setFieldValue( y1Instance , hole1CenterYField );
}


// Hole 2 diameter value
if( d2Instance.isOutput() )
{
    APMRealInstance d2Instance = plateInstance.
    getObjectInstance( "hole2" ).
    getRealInstance( "diameter" );
    setFieldValue( d2Instance , hole2DiameterField );
}


// Hole 2 center X value
if( x2Instance.isOutput() )
{
    APMRealInstance x2Instance = plateInstance.
    getObjectInstance( "hole2" ).
    getObjectInstance( "center" ).
    getRealInstance( "x" );
    setFieldValue( x2Instance , hole2CenterXField );
}
```

```
    // Hole 2 center Y value
    if( y2Instance.isOutput() )
    {
        APMRealInstance y2Instance = plateInstance.
        getObjectInstance( "hole2" ).
        getObjectInstance( "center" ).
        getRealInstance( "y" );
        setFieldValue( y2Instance , hole2CenterYField );
    }



    // Check the isInput checkbox if the variable is an input
    designerIsInputCheckbox.setState( designerInstance.isInput() );
    materialIsInputCheckbox.setState( materialInstance.isInput() );
    youngsModulusIsInputCheckbox.setState( youngsModulusInstance.isInput() );
    L1IsInputCheckbox.setState( L1Instance.isInput() );
    L2IsInputCheckbox.setState( L2Instance.isInput() );
    L3IsInputCheckbox.setState( L3Instance.isInput() );
    lengthIsInputCheckbox.setState( lengthInstance.isInput() );
    widthIsInputCheckbox.setState( widthInstance.isInput() );
    thicknessIsInputCheckbox.setState( thicknessInstance.isInput() );
    hole1DiameterIsInputCheckbox.setState( d1Instance.isInput() );
    hole1CenterXIsInputCheckbox.setState( x1Instance.isInput() );
    hole1CenterYIsInputCheckbox.setState( y1Instance.isInput() );
    hole2DiameterIsInputCheckbox.setState( d2Instance.isInput() );
    hole2CenterXIsInputCheckbox.setState( x2Instance.isInput() );
    hole2CenterYIsInputCheckbox.setState( y2Instance.isInput() );
    criticalAreaIsInputCheckbox.setState( criticalAreaInstance.isInput() );


}

private void setFieldValue( APMRealInstance instance , TextField textField )
{
    int numberOfSolutions;

    if( instance.hasValue() )
    {
        double value = instance.getRealValue();
        textField.setText( Double.toString( value ) );
    }
    else
    {
        numberOfSolutions = instance.trySolveForValue();
        if( numberOfSolutions > 0 )
        {
            double value = instance.getRealValue();
            textField.setText( Double.toString( value ) );
        }
        else
        textField.setText( "" );
    }
}


private void colorAttributeNameLabels()
{
    APMObjectDomain plateDomain = (APMObjectDomain) APMInterface.getAPMDomain( "back_plate_geometric_model" ,
"plate" );
    APMPrimitiveAttribute attribute;

    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "designer.first_name" ) );
    if( attribute.isEssentialAttribute() )
        designerLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        designerLabel.setForeground( Color.red );
```

```
else if( attribute.isProductAttribute() )
    designerLabel.setForeground( Color.darkGray );

attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "material.materialName" ) );
if( attribute.isEssentialAttribute() )
    materialLabel.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    materialLabel.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    materialLabel.setForeground( Color.darkGray );

attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "material.youngsModulus" ) );
if( attribute.isEssentialAttribute() )
    youngsModulusLabel.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    youngsModulusLabel.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    youngsModulusLabel.setForeground( Color.darkGray );


attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "l1" ) );
if( attribute.isEssentialAttribute() )
    L1Label.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    L1Label.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    L1Label.setForeground( Color.darkGray );


attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "l2" ) );
if( attribute.isEssentialAttribute() )
    L2Label.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    L2Label.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    L2Label.setForeground( Color.darkGray );


attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "l3" ) );
if( attribute.isEssentialAttribute() )
    L3Label.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    L3Label.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    L3Label.setForeground( Color.darkGray );

attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "length" ) );
if( attribute.isEssentialAttribute() )
    lengthLabel.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    lengthLabel.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    lengthLabel.setForeground( Color.darkGray );

attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "thickness" ) );
if( attribute.isEssentialAttribute() )
    thicknessLabel.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    thicknessLabel.setForeground( Color.red );
else if( attribute.isProductAttribute() )
    thicknessLabel.setForeground( Color.darkGray );

attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "width" ) );
if( attribute.isEssentialAttribute() )
    widthLabel.setForeground( Color.blue );
else if( attribute.isIdealizedAttribute() )
    widthLabel.setForeground( Color.red );
else if( attribute.isProductAttribute() )
```

```
        widthLabel.setForeground( Color.darkGray );

    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole1.diameter" ) );
    if( attribute.isEssentialAttribute() )
        hole1DiameterLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole1DiameterLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole1DiameterLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole1.center.x" ) );
    if( attribute.isEssentialAttribute() )
        hole1CenterXLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole1CenterXLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole1CenterXLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole1.center.y" ) );
    if( attribute.isEssentialAttribute() )
        hole1CenterYLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole1CenterYLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole1CenterYLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole2.diameter" ) );
    if( attribute.isEssentialAttribute() )
        hole2DiameterLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole2DiameterLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole2DiameterLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole2.center.x" ) );
    if( attribute.isEssentialAttribute() )
        hole2CenterXLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole2CenterXLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole2CenterXLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "hole2.center.y" ) );
    if( attribute.isEssentialAttribute() )
        hole2CenterYLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        hole2CenterYLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        hole2CenterYLabel.setForeground( Color.darkGray );


    attribute = (APMPrimitiveAttribute) plateDomain.getAttribute( new ListOfStrings( "critical_area" ) );
    if( attribute.isEssentialAttribute() )
        criticalAreaLabel.setForeground( Color.blue );
    else if( attribute.isIdealizedAttribute() )
        criticalAreaLabel.setForeground( Color.red );
    else if( attribute.isProductAttribute() )
        criticalAreaLabel.setForeground( Color.darkGray );

}
```

```
    private void addComponent( Component component , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        variablesPanelGridbagConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        variablesPanelGridbagConstraints.gridx = column;
        variablesPanelGridbagConstraints.gridy = row;

        // Set gridwidth and gridheight
        variablesPanelGridbagConstraints.gridwidth = width;
        variablesPanelGridbagConstraints.gridheight = height;

        // Set constraints
        variablesPanelGridbagLayout.setConstraints( component , variablesPanelGridbagConstraints );
        add( component );
    }


}
```

AnalysisPanel.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;
import apm.solver.*;
import gui.*;

public class AnalysisPanel extends Panel implements ActionListener , ItemListener
{
    private GridBagConstraints analysisPanelGridbagConstraints;
    private GridBagLayout analysisPanelGridbagLayout;
    private MainFrame mainFrame;

    private Label inputLabel;
    private Label loadLabel;
    private Label stressLabel;
    private Label elongationLabel;
    private TextField loadField;
    private TextField stressField;
    private TextField elongationField;
    private Checkbox loadIsInputCheckbox;
    private Checkbox stressIsInputCheckbox;
    private Checkbox elongationIsInputCheckbox;

    private Button runAnalysisButton;


    public AnalysisPanel( MainFrame m )
    {
        mainFrame = m;

        // Set the layout manager for the panel (a GridLayoutBag)
        analysisPanelGridbagLayout = new GridBagLayout();
        analysisPanelGridbagConstraints = new GridBagConstraints();
        setLayout( analysisPanelGridbagLayout );


        int row = 0 ;

        // Input label
        inputLabel = new Label( "Input" );
        inputLabel.setAlignment( Label.CENTER );
        addComponent( inputLabel , row , 2 , 1 , 1 , 0 , 0 , 0 , 0 );

        // Load row
        loadLabel = new Label( "Load" );
```

```
    loadLabel.setAlignment( Label.RIGHT );
    addComponent( loadLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    loadField = new TextField( 20 );
    loadField.setBackground( Color.lightGray );
    addComponent( loadField , row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );
    loadIsInputCheckbox = new Checkbox( );
    addComponent( loadIsInputCheckbox , row , 2 , 1 , 1 , 0 , 10 , 0 , 5 );

    // Stress row
    stressLabel = new Label( "Stress" );
    stressLabel.setAlignment( Label.RIGHT );
    addComponent( stressLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    stressField = new TextField( 20 );
    stressField.setBackground( Color.lightGray );
    addComponent( stressField , row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );
    stressIsInputCheckbox = new Checkbox( );
    addComponent( stressIsInputCheckbox , row , 2 , 1 , 1 , 0 , 10 , 0 , 5 );


    // Elongation row
    elongationLabel = new Label( "Elongation" );
    elongationLabel.setAlignment( Label.RIGHT );
    addComponent( elongationLabel , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
    elongationField = new TextField( 20 );
    elongationField.setBackground( Color.lightGray );
    addComponent( elongationField , row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );
    elongationIsInputCheckbox = new Checkbox( );
    addComponent( elongationIsInputCheckbox , row , 2 , 1 , 1 , 0 , 10 , 0 , 5 );


    // Calculate button
    runAnalysisButton = new Button( "Run Analysis" );
    addComponent( runAnalysisButton , ++row , 1 , 1 , 1 , 20 , 10 , 20 , 0 );


    // Add action listeners to value fields to detect a change in value
    loadField.addActionListener( this );
    stressField.addActionListener( this );
    elongationField.addActionListener( this );


    // Add item listeners to check boxes
    loadIsInputCheckbox.addItemListener( this );
    stressIsInputCheckbox.addItemListener( this );
    elongationIsInputCheckbox.addItemListener( this );

    // Add action listener to solve button
    runAnalysisButton.addActionListener( this );


}

public void paint( Graphics g )
{
    /*
    g.setFont( new Font( "Courier" , Font.BOLD , 20 ) );
    g.setColor( Color.blue );
    g.drawRect( 5 , 25 , 300 , 150 );
    g.drawString( "Tension Analysis" , 65 , 20 );
    */
}

public void actionPerformed( ActionEvent e )
{
    // Calculate button has been pressed: Perform analysis
    if( e.getSource() == runAnalysisButton )
    {
        String relation1 = "stress == load/criticalArea";
```

```
String relation2 = "elongation == ( load * length )/( criticalArea * youngsModulus )";
ListOfStrings listOfRelations = new ListOfStrings();
listOfRelations.addElement( relation1 );
listOfRelations.addElement( relation2 );
ListOfStrings listOfInputVariableNames = new ListOfStrings();
ListOfReals listOfInputValues = new ListOfReals();


// Create a list of inputs for the analysis
// For analysis, inputs are: 1) ANY APM variable with value and 2) INPUT Analysis variables
if( loadIsInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "load" );
    listOfInputValues.addElement( Double.valueOf( loadField.getText() ).doubleValue() );
}
if( stressIsInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "stress" );
    listOfInputValues.addElement( Double.valueOf( stressField.getText() ).doubleValue() );
}
if( elongationIsInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "elongation" );
    listOfInputValues.addElement( Double.valueOf( elongationField.getText() ).doubleValue() );
}
if( mainFrame.variablesPanel.criticalAreaIsAnalysisInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "criticalArea" );
    listOfInputValues.addElement( Double.valueOf( mainFrame.variablesPanel.criticalAreaField.getText() ).doubleValue() );
}
if( mainFrame.variablesPanel.youngsModulusIsAnalysisInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "youngsModulus" );
    listOfInputValues.addElement( Double.valueOf( mainFrame.variablesPanel.youngsModulusField.getText() ).doubleValue() );
}
if( mainFrame.variablesPanel.lengthIsAnalysisInputCheckbox.getState() == true )
{
    listOfInputVariableNames.addElement( "length" );
    listOfInputValues.addElement( Double.valueOf( mainFrame.variablesPanel.lengthField.getText() ).doubleValue() );
}


// Solve for the analysis outputs
APMSolverWrapper solver = APMSolverWrapperFactory.makeSolverWrapperFor( "mathematica" );
APMSolverResult solverResults;

if( loadIsInputCheckbox.getState() == false )
{
    solverResults = solver.solveFor( "load" , listOfRelations , listOfInputVariableNames , listOfInputValues );
    loadField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
}

if( stressIsInputCheckbox.getState() == false )
{
    solverResults = solver.solveFor( "stress" , listOfRelations , listOfInputVariableNames , listOfInputValues );
    stressField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
}
if( elongationIsInputCheckbox.getState() == false )
{
    solverResults = solver.solveFor( "elongation" , listOfRelations , listOfInputVariableNames , listOfInputValues );
    elongationField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
}
if( mainFrame.variablesPanel.criticalAreaIsAnalysisInputCheckbox.getState() == false )
{
    solverResults = solver.solveFor( "criticalArea" , listOfRelations , listOfInputVariableNames , listOfInputValues );
    mainFrame.variablesPanel.criticalAreaField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
}
if( mainFrame.variablesPanel.youngsModulusIsAnalysisInputCheckbox.getState() == false)
```

```
            {
                solverResults = solver.solveFor( "youngsModulus" , listOfRelations , listOfInputVariableNames , listOfInputValues );
                mainFrame.variablesPanel.youngsModulusField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
            }
            if( mainFrame.variablesPanel.lengthIsAnalysisInputCheckbox.getState() == false )
            {
                solverResults = solver.solveFor( "length" , listOfRelations , listOfInputVariableNames , listOfInputValues );
                mainFrame.variablesPanel.lengthField.setText( String.valueOf( solverResults.getResults().elementAt( 0 ) ) );
            }


        }

    }


    public void itemStateChanged( ItemEvent e )
    {
        // An isInput checkbox has been clicked.
        if( e.getSource() == loadIsInputCheckbox )
            loadField.setEditable( loadIsInputCheckbox.getState() );
        else if( e.getSource() == stressIsInputCheckbox )
            stressField.setEditable( stressIsInputCheckbox.getState() );
        else if( e.getSource() == elongationIsInputCheckbox )
            elongationField.setEditable( elongationIsInputCheckbox.getState() );


    }


    private void addComponent( Component component , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        analysisPanelGridbagConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        analysisPanelGridbagConstraints.gridx = column;
        analysisPanelGridbagConstraints.gridy = row;

        // Set gridwidth and gridheight
        analysisPanelGridbagConstraints.gridwidth = width;
        analysisPanelGridbagConstraints.gridheight = height;

        // Set constraints
        analysisPanelGridbagLayout.setConstraints( component , analysisPanelGridbagConstraints );
        add( component );
    }

}
```

RelationsPanel.java

```
import java.awt.*;
import java.awt.event.*;
import apm.*;
import constraint.*;

public class RelationsPanel extends Panel implements ItemListener
{
    private GridBagLayout gbLayout;
    private GridBagConstraints gbConstraints;
    private MainFrame mainFrame;
    private ListOfConstraintNetworkRelations relations;
    private Label tempRelationNameLabel[] = new Label[ 10 ];
    private TextField tempRelationExpressionField[] = new TextField[ 10 ];
    private Checkbox tempRelationIsActiveCheckbox[] = new Checkbox[ 10 ];

    public RelationsPanel( MainFrame m )
```

```
{
    ConstraintNetworkRelation tempRelation;
    String tempRelationName;
    String tempRelationExpression;

    mainFrame = m;

    // Set the layout manager for the panel (a GridLayoutBag)
    gbLayout = new GridBagLayout();
    gbConstraints = new GridBagConstraints();
    setLayout( gbLayout );


    ConstraintNetwork constraintNetwork = APMInterface.getConstraintNetwork();

    relations = constraintNetwork.getRelations();

    int row = 0;

    Label isActiveLabel = new Label( "Active" );
    addComponent( isActiveLabel , row , 2 , 1 , 1 , 0 , 10 , 0 , 0 );

    for( int i = 0 ; i < relations.size() ; i++ )
    {
        tempRelation = relations.elementAt( i );
        tempRelationName = tempRelation.getName();
        tempRelationExpression = tempRelation.getExpression();

        tempRelationNameLabel[ i ] = new Label( tempRelationName );
        tempRelationNameLabel[ i ].setAlignment( Label.RIGHT );

        // Color
        if( tempRelation.getCategory() == APMRelation.PRODUCT_RELATION )
            tempRelationNameLabel[ i ].setForeground( Color.blue );
        else if( tempRelation.getCategory() == APMRelation.PRODUCT_IDEALIZATION_RELATION )
            tempRelationNameLabel[ i ].setForeground( Color.red );
        addComponent( tempRelationNameLabel[ i ] , ++row , 0 , 1 , 1 , 0 , 10 , 0 , 0 );
        tempRelationExpressionField[ i ] = new TextField( 30 );
        tempRelationExpressionField[ i ].setText( tempRelationExpression );
        tempRelationExpressionField[ i ].setEditable( false );
        tempRelationExpressionField[ i ].setBackground( Color.lightGray );
        addComponent( tempRelationExpressionField[ i ] , row , 1 , 1 , 1 , 0 , 10 , 0 , 0 );
        tempRelationIsActiveCheckbox[ i ] = new Checkbox();
        addComponent( tempRelationIsActiveCheckbox[ i ] , row , 2 , 1 , 1 , 0 , 10 , 0 , 0 );
        tempRelationIsActiveCheckbox[ i ].setState( tempRelation.isActive() );
        tempRelationIsActiveCheckbox[ i ].addItemListener( this );

    }
}

public void itemStateChanged( ItemEvent e )
{
    ListOfConstraintNetworkVariables listOfConnectedVariables;

    // Get the list of variables connected to the relation whose active status changed
    for( int i = 0 ; i < tempRelationIsActiveCheckbox.length ; i++ )
        if( e.getSource() == tempRelationIsActiveCheckbox[ i ] )
        {
            // Switch the active flag
            relations.elementAt( i ).setActive( tempRelationIsActiveCheckbox[ i ].getState() );
        }


}
```

```
    private void addComponent( Component component , int row , int column , int width , int height ,
    int spaceTop , int spaceLeft , int spaceBottom , int spaceRight )
    {
        gbConstraints.insets = new Insets( spaceTop , spaceLeft , spaceBottom , spaceRight );
        // Set gridx and gridy
        gbConstraints.gridx = column;
        gbConstraints.gridy = row;

        // Set gridwidth and gridheight
        gbConstraints.gridwidth = width;
        gbConstraints.gridheight = height;

        // Set constraints
        gbLayout.setConstraints( component , gbConstraints );
        add( component );
    }


}
```

II.4 APM Browser

APMBrowser.java

```
import gui.CloseWindowAndExit;

public class APMBrowser
{
    public static void main( String args[] )
    {
        APMBrowserFrame apmBrowserFrame = new APMBrowserFrame();
        apmBrowserFrame.addWindowListener( new CloseWindowAndExit() );
    }
```

APMBrowserFrame.java

```
    import java.awt.*;
    import java.awt.event.*;
    import apm.*;
    import gui.*;

    public class APMBrowserFrame extends Frame implements ActionListener
    {
        private TextArea t;
        private MenuBar bar;
        private Menu fileMenu;

        private MenuItem loadAPMDefinitions;
        private MenuItem loadSourceData;
        private Menu saveAPMSubMenu;
        private MenuItem saveLinkedAPMDefinition;
        private MenuItem exportToExpress;
        private Menu saveDataSubMenu;
        private MenuItem saveInstancesBySourceSet;
        private MenuItem saveLinkedInstances;
        private MenuItem exitBrowser;
        private Menu apmStructureMenu;
        private Menu printAPMSubMenu;
        private Menu printUnlinkedAPMSubMenu;
        private MenuItem printUnlinkedAPMDefinitionsToScreen;
        private MenuItem printUnlinkedAPMDefinitionsToFile;
        private Menu printLinkedAPMSubMenu;
        private MenuItem printLinkedAPMDefinitionsToScreen;
        private MenuItem printLinkedAPMDefinitionsToFile;
        private Menu dataMenu;
        private Menu printDataSubMenu;
        private Menu printUnlinkedDataSubMenu;
        private MenuItem printUnlinkedAPMInstancesToScreen;
        private MenuItem printUnlinkedAPMInstancesToFile;
        private Menu printLinkedDataSubMenu;
        private MenuItem printLinkedAPMInstancesToScreen;
        private MenuItem printLinkedAPMInstancesToFile;

        private String lastDirectoryName;
        private String apmDefinitionsFileName;


        public APMBrowserFrame()
        {
            super( "Analizable Product Model Browser" );
            setSize( 500 , 550 );

            t = new TextArea( " " , 2 , 20 , TextArea.SCROLLBARS_BOTH );
            t.setEditable( false );
            add( t , BorderLayout.CENTER );

            // Create menubar
```

```
bar = new MenuBar();

// File menu
fileMenu = new Menu( "File" );
loadAPMDefinitions = new MenuItem( "Load APM Definition" );
loadSourceData = new MenuItem( "Load Source Data" );
saveAPMSubMenu = new Menu( "Save APM" );
saveLinkedAPMDefinition = new MenuItem( "Linked" );
exportToExpress = new MenuItem( "EXPRESS" );
saveDataSubMenu = new Menu( "Save Data" );
saveInstancesBySourceSet = new MenuItem( "By Source Set" );
saveLinkedInstances = new MenuItem( "Linked" );
exitBrowser = new MenuItem( "Exit" );

saveAPMSubMenu.add( saveLinkedAPMDefinition );
saveAPMSubMenu.add( exportToExpress );
saveDataSubMenu.add( saveInstancesBySourceSet );
saveDataSubMenu.add( saveLinkedInstances );
fileMenu.add( loadAPMDefinitions );
fileMenu.add( loadSourceData );
fileMenu.addSeparator();
fileMenu.add( saveAPMSubMenu );
fileMenu.add( saveDataSubMenu );
fileMenu.addSeparator();
fileMenu.add( exitBrowser );

loadAPMDefinitions.addActionListener( this );
loadSourceData.addActionListener( this );
saveLinkedAPMDefinition.addActionListener( this );
exportToExpress.addActionListener( this );
saveInstancesBySourceSet.addActionListener( this );
saveLinkedInstances.addActionListener( this );
exitBrowser.addActionListener( this );


// APM Structure Menu
apmStructureMenu = new Menu( "APM Structure" );
printAPMSubMenu = new Menu( "Print APM" );
printUnlinkedAPMSubMenu = new Menu( "Unlinked" );
printUnlinkedAPMDefinitionsToScreen = new MenuItem( "To screen" );
printUnlinkedAPMDefinitionsToFile = new MenuItem( "To file" );
printLinkedAPMSubMenu = new Menu( "Linked" );
printLinkedAPMDefinitionsToScreen = new MenuItem( "To screen" );
printLinkedAPMDefinitionsToFile = new MenuItem( "To file" );

printUnlinkedAPMSubMenu.add( printUnlinkedAPMDefinitionsToScreen );
printUnlinkedAPMSubMenu.add( printUnlinkedAPMDefinitionsToFile );
printLinkedAPMSubMenu.add( printLinkedAPMDefinitionsToScreen );
printLinkedAPMSubMenu.add( printLinkedAPMDefinitionsToFile );
printAPMSubMenu.add( printUnlinkedAPMSubMenu );
printAPMSubMenu.add( printLinkedAPMSubMenu );
apmStructureMenu.add( printAPMSubMenu );

printUnlinkedAPMDefinitionsToScreen.addActionListener( this );
printUnlinkedAPMDefinitionsToFile.addActionListener( this );
printLinkedAPMDefinitionsToScreen.addActionListener( this );
printLinkedAPMDefinitionsToFile.addActionListener( this );


// Data Menu
dataMenu = new Menu( "Data" );
printDataSubMenu = new Menu( "Print Data" );
printUnlinkedDataSubMenu = new Menu( "Unlinked" );
printUnlinkedAPMInstancesToScreen = new MenuItem( "To screen" );
printUnlinkedAPMInstancesToFile = new MenuItem( "To file" );
printLinkedDataSubMenu = new Menu( "Linked" );
printLinkedAPMInstancesToScreen = new MenuItem( "To screen" );
printLinkedAPMInstancesToFile = new MenuItem( "To file" );
```

```
        printUnlinkedDataSubMenu.add( printUnlinkedAPMInstancesToScreen );
        printUnlinkedDataSubMenu.add( printUnlinkedAPMInstancesToFile );
        printLinkedDataSubMenu.add( printLinkedAPMInstancesToScreen );
        printLinkedDataSubMenu.add( printLinkedAPMInstancesToFile );
        printDataSubMenu.add( printUnlinkedDataSubMenu );
        printDataSubMenu.add( printLinkedDataSubMenu );
        dataMenu.add( printDataSubMenu );

        printUnlinkedAPMInstancesToScreen.addActionListener( this );
        printUnlinkedAPMInstancesToFile.addActionListener( this );
        printLinkedAPMInstancesToScreen.addActionListener( this );
        printLinkedAPMInstancesToFile.addActionListener( this );



        // Add menus to the bar
        bar.add( fileMenu );
        bar.add( apmStructureMenu );
        bar.add( dataMenu );

        // Add bar
        setMenuBar( bar );

        setVisible( true );


    }


    public void actionPerformed( ActionEvent e )
    {
        t.setText( "" );

        boolean success = false;
        FileDialog fileDialog;
        InfoDialog infoDialog;

        if( e.getSource() == loadAPMDefinitions )
        {
            // Create file dialog and get the file name
            fileDialog = new FileDialog( this , "Select APM Definition File" , FileDialog.LOAD );
            fileDialog.show();
            lastDirectoryName = fileDialog.getDirectory();
            apmDefinitionsFileName = lastDirectoryName + fileDialog.getFile();


            // Initialize the interface
            APMInterface.initialize();

            success = APMInterface.loadAPMDefinitions( apmDefinitionsFileName );

            // Display a dialog notifying wheter or not the APM definitions have been loaded
            if( success )
                infoDialog = new InfoDialog( this , "Message" , "APM loaded successfully" );
            else
                infoDialog = new InfoDialog( this , "Message" , "APM not loaded" );

            infoDialog.show();
        }

        else if( e.getSource() == loadSourceData )
        {
            APMSourceSet sourceSetCursor;
            ListOfStrings listOfFileNames = new ListOfStrings();

            // Prompt user for a data file for each source set
            for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
```

```
    {
        sourceSetCursor = APMInterface.getSourceSets().elementAt( i );

        fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
    FileDialog.LOAD );
        fileDialog.setDirectory( lastDirectoryName );
        fileDialog.show();
        listOfFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
    }

    // Load the data
    success = APMInterface.loadSourceSetData( listOfFileNames );


    // Display a message indicating wheter or not the data was loaded succesfully
    if( success )
        infoDialog = new InfoDialog( this , "Message" , "Data loaded successfully" );
    else
        infoDialog = new InfoDialog( this , "Message" , "Data not loaded" );

    infoDialog.show();


}

else if( e.getSource() == saveLinkedAPMDefinition )
{
    fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
    fileDialog.setDirectory( lastDirectoryName );
    fileDialog.show();
    APMInterface.saveLinkedAPMDefinition( fileDialog.getDirectory() + fileDialog.getFile() );
}

else if( e.getSource() == exportToExpress )
APMInterface.exportToExpress( lastDirectoryName );


else if( e.getSource() == saveInstancesBySourceSet )
{
    APMSourceSet sourceSetCursor;
    ListOfStrings listOfOutputFileNames = new ListOfStrings();

    // Prompt user for a data file for each source set
    for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
    {
        sourceSetCursor = APMInterface.getSourceSets().elementAt( i );
        fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
    FileDialog.LOAD );
        fileDialog.setDirectory( lastDirectoryName );
        fileDialog.show();
        listOfOutputFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
    }

    APMInterface.saveInstancesBySourceSet( listOfOutputFileNames );
}

else if( e.getSource() == saveLinkedInstances )
{
    fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
    fileDialog.setDirectory( lastDirectoryName );
    fileDialog.show();
    APMInterface.saveLinkedInstances( fileDialog.getDirectory() + fileDialog.getFile() );
}

else if( e.getSource() == exitBrowser )
{
    System.exit( 0 );
}
```

```
        else if( e.getSource() == printUnlinkedAPMDefinitionsToScreen )
            t.setText( APMInterface.printUnlinkedAPMDefinitions( ) );

        else if( e.getSource() == printUnlinkedAPMDefinitionsToFile )
        {
            fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
            fileDialog.setDirectory( lastDirectoryName );
            fileDialog.show();
            APMInterface.printUnlinkedAPMDefinitions( fileDialog.getDirectory() + fileDialog.getFile() );
        }

        else if( e.getSource() == printLinkedAPMDefinitionsToScreen )
            t.setText( APMInterface.printLinkedAPMDefinitions( ) );


        else if( e.getSource() == printLinkedAPMDefinitionsToFile )
        {
            fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
            fileDialog.setDirectory( lastDirectoryName );
            fileDialog.show();
            APMInterface.printLinkedAPMDefinitions( fileDialog.getDirectory() + fileDialog.getFile() );
        }


        else if( e.getSource() == printUnlinkedAPMInstancesToScreen )
            t.setText( APMInterface.printUnlinkedAPMInstances( ) );

        else if( e.getSource() == printUnlinkedAPMInstancesToFile )
        {
            APMSourceSet sourceSetCursor;
            ListOfStrings listOfOutputFileNames = new ListOfStrings();

            // Prompt user for a data file for each source set
            for( int i = 0 ; i < APMInterface.getSourceSets().size() ; i++ )
            {
                sourceSetCursor = APMInterface.getSourceSets().elementAt( i );
                fileDialog = new FileDialog( this , "Select Data File for: \"" + sourceSetCursor.getSourceSetName() + "\"" ,
            FileDialog.LOAD );
                fileDialog.setDirectory( lastDirectoryName );
                fileDialog.show();
                listOfOutputFileNames.addElement( fileDialog.getDirectory() + fileDialog.getFile() );
            }

            APMInterface.printUnlinkedAPMInstances( listOfOutputFileNames );
        }


        else if( e.getSource() == printLinkedAPMInstancesToScreen )
            t.setText( APMInterface.printLinkedAPMInstances() );

        else if( e.getSource() == printLinkedAPMInstancesToFile )
        {
            fileDialog = new FileDialog( this , "Save File:" , FileDialog.SAVE );
            fileDialog.setDirectory( lastDirectoryName );
            fileDialog.show();
            APMInterface.printLinkedAPMInstances( fileDialog.getDirectory() + fileDialog.getFile() );
        }

    }

}
```

VITA

Diego Romano Tamburini was born in Caracas, Venezuela on August 9, 1966 to Franca and Libero Tamburini, both Italian immigrants who had been living in Maracay (a city at approximately 150 kilometers south of Caracas) since the early fifties.

Mr. Tamburini spent his entire childhood and adolescence in Maracay, where he attended Maracay's Marist School throughout his elementary and secondary years. He graduated from high school on July of 1983 and started his university education in September of the same year at the Polytechnic University Institute of the Venezuelan Armed Forces (IUPFAN – from the initials in Spanish) in Maracay, where he received military training in conjunction with his Mechanical Engineering instruction. In the fall of 1985 he transferred to IUPFAN's Caracas campus to complete the final quarters of his curriculum. He was an intern for two academic terms at the Venezuelan Institute of Oil Technology (INTEVEP) where he developed his undergraduate thesis. He received his degree in Mechanical Engineering on December of 1987.

From January of 1988 until August of the same year he worked in Maracay as an Assistant Engineer with the Numerical Controlled Machining Division of the Venezuelan Military Industries. In September of 1988, he started his graduate studies at the Georgia Institute of Technology in Atlanta, Georgia (United States of America), where he received his Masters Degree in Mechanical Engineering in December of 1989.

After receiving his masters he joined the Design Department of Lagoven, an affiliate of Venezuelan Oil Company in Maracaibo (the second largest city in Venezuela, approximately 600 kilometers west of Caracas, where the largest portion of Venezuela's oil industry is based) as a CAD/CAE Project Engineer, where he worked from January of 1990 until December of 1992. In Lagoven he met Patricia Esparza, whom he married in December of 1991.

In January of 1993 he began his doctoral studies in Mechanical Engineering also at the Georgia Institute of Technology which he will complete, with this thesis, in the Spring quarter of 1999.

In September of 1998 - still during the final stages of writing this thesis - Mr. Tamburini joined the Metaphase Division of SDRC, where he currently works as a Senior Product Data Management Implementation Engineer, in Seattle, Washington.