BUILDING PRODUCT MODELS

Computer Environments Supporting Design and Construction

Charles M. Eastman

Georgia Institute of Technology
Atlanta GA. USA

CRC Press
Boca Raton FL

CHAPTER FIVE
ISO-STEP

...the opportunity to develop EXPRESS was a chance to improve upon what seemed to be an inadequate way of thinking about and documenting what we know about information, which influences our lives so greatly. We are pushed into either of two corners: one that dealt with data and relationships only and another that entangled information with every conceivable computer application development detail. From my point of view, information is certainly more than the former and definitely should be kept apart from the latter.

Douglas Schenck and Peter Wilson
Preface
Information Modeling the EXPRESS Way
1994

5.1 INTRODUCTION

By 1984, the issues of CAD data translation, as summarized at the end of Chapter Three, suggested to a number of industry-based research groups in Europe and the US that the time was appropriate for a new generation of standards efforts. In the US, the new effort was centered around the Product Data Exchange Standard (PDES). About the same time, the International Standards Organization (ISO) in Geneva, Switzerland, initiated a Technical Committee, TC184, to initiate a subcommittee, SC4, to develop a standard called STEP (STandard for the Exchange of Product Model Data). The full title of the standard is *ISO1303 - Industrial Automation Systems - Product Data Representation and Exchange*. The STEP effort was initiated in part because different European countries were embarking on development of their own standards. Beside IGES and PDES, there was SET by the French, CAD*I by the Germans, and other efforts such as VDA-FS and EDIF. After initially operating as parallel but separate activities, the PDES and STEP efforts merged in 1991. Today, the international committees working on STEP meet quarterly, twice a year in the USA, once a year in Europe, and once a year in Asia.

This chapter reviews the overall structure of ISO-STEP and its approach to data exchange. It surveys the various languages—specifically NIAM, EXPRESS and EXPRESS-G—that are used in developing specifications within STEP and also some of the tools available for implementing STEP-based translators. In later chapters, we review additional facilities surrounding STEP and some of the exchange models developed with it. (The reader is warned that these international standards efforts use many acronyms and are a veritable "alphabet soup".)

The primary motivation for these new efforts in data exchange was the recognition that IGES and similar efforts had some basic weaknesses that were not easily corrected. The new effort had objectives to

- incorporate new programming language concepts, especially those dealing with object-oriented programming
- incorporate formal specifications of the structures defined, using the new recently developed data modeling languages
- separate the data model from the physical file format

- support subsets of a total model, allowing clusters of applications to be integrated without the overhead of having to deal with parts of a model irrelevant to a task
- support alternative physical level implementations, including files, databases and knowledge-based systems
- incorporate reference models that are common shared subsets of larger standard models

5.2 THE STRUCTURE OF ISO-STEP

An influence on STEP thinking was the work of the American National Standards Institute (ANSI-SPARC) committee on database architectures and standards. This work distinguished between database definition and implementation, defining a layered architecture of abstractions and mappings. The layered architecture consisted of three levels:

> *Physical*: the physical implementation of the conceptual schema on a file system; this level of definition defines the physical structure of data on disk or other media.
>
> *Logical*: the information of interest, defined in a logical structure; this level of specification presents, in an implementation-independent manner, the logical structure of data and the relationships it holds; the logical level maps to the physical level.
>
> *Application*: the information subset relevant to a particular application (later called a view); this level of definition specifies the information needed by a specific application and its format; the application level was implemented on top of and was derived from the logical level.

These levels distinguish the logical structure of the information from the format in which it is carried on some medium. This separation was a fundamental idea in the new STEP approach, as we shall see. It also separates an application level that is written in or on top of the logical level, suggesting a way to define subsets of a complete model.

The STEP Committee decided at the outset to use information-modeling methods to specify the required conceptual structure of the information to be represented. They accepted two information models as tools to formally specify the conceptual requirements—IDEF1x and NIAM. IDEF1x had been developed for the definition of defense planning in the US and NIAM was developed for business database schema design in Europe. The only popular information model not included was the original Entity-Relationship model (ER) and its extensions. Later, they also added EXPRESS-G. In addition, they approved development of an intermediate-level specification language for defining the logical structure of a model, separate from its physical implementation. For this, they commissioned development of the EXPRESS language.

Rather than define a complete information model and then take subsets of it, the STEP architecture started instead by defining various part models, called Application Protocols (APs), with the expectation that they would later be reconciled into larger domain-specific Framework models. In this respect, the STEP approach has been very different from IGES. In the building context, this means that APs may be defined for structural steel, reinforced concrete, curtainwalls, mechanical systems and so forth. A complete building model would logically come only after the development of some sets of APs. This seems to have been based on the recognition that existing applications with rich semantic content are clustered into such specialized domains. Another motivation of the AP approach was the need to generate useful, incremental results quickly.

How are these various ideas organized into a data exchange system? The STEP system architecture identifies five classes of tools. They are diagrammed in Figure 5.1 with the information flows between various tools. The five classes of tools are:

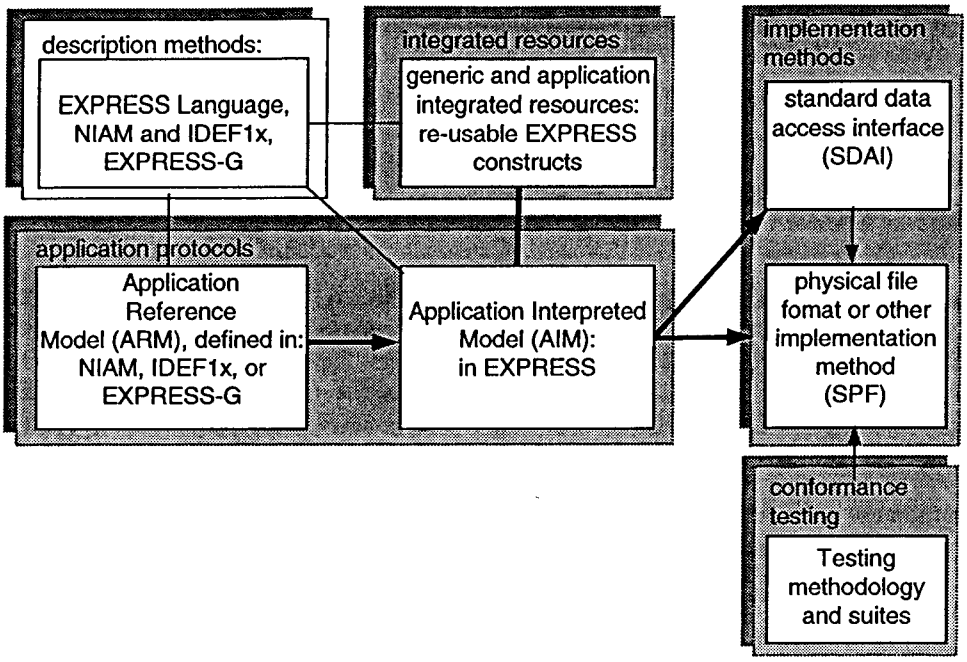

Figure 5.1: A diagrammatic representation of the different Parts of STEP, giving their names and how they are used. The thin lines designate language use, while the heavier arrows indicate a mapping realized by a translator. The one heavy line without an arrow indicates reuse of existing models.

1. A data exchange system utilizes various *description methods*, which are the information-modeling languages employed in specifying the information models used in the architecture, i.e., to define the Integrated Resources, Application Reference Models and Application Interpreted Models. The formal description methods include NIAM, IDEF1x, EXPRESS and EXPRESS-G.

2. *Integrated resources* are the common model subsets that get used repeatedly in the definition of application models. Models used in different domains are called *generic* integrated resources and include geometry, material properties and project classifications—that is, items that can be shared across multiple application domains. Model subsets that are industry specific are called *application* integrated resources. These include subsets for electronics, drafting, kinematics, finite elements and building. Presentation formats are called *constructs.*

3. *Application protocols* (APs) are developed for particular application contexts, using the Description Methods and Integrated Resources. An application protocol is partitioned into two aspects: an *Application Reference Model (ARM)* and an *Application Interpreted Model (AIM).* An Application Reference Model represents requirements for an application in a form easily understood by knowledgeable users, for designing and assessing the information model. NIAM, IDEF1x and EXPRESS-G are initially used as languages for defining ARMs. An Application Reference Model is interpreted into an Application Interpreted Model, which is readable by both people and computers. EXPRESS is the language used to define AIMs. The AIM resolves all uses of the Generic Integrated Resources and integrates the model with Application Integrated Resources.

4. An application protocol is combined with an *implementation method* to form the basis for a STEP implementation. An implementation method typically includes multiple resources. The

STEP physical file (SPF) and the Standard Data Access Interface (SDAI) to the SPF are the Implementation Methods that have been developed thus far.

5. Finally, each STEP application protocol and implementation requires *conformance testing*. A conformance test assesses the implementation in terms of its ARM and AIM and confirms that the STEP languages and tools have been properly used and interpreted. A Conformance Testing methodology is applied by accredited organizations. The testing includes application and interpretation of test suites.

The components of an AP include an Application Reference Model, an Application Interpreted Model and Conformance Testing requirements. Together, these provide the specific functionality for an application requirement in a self-contained form. The ARM states the needs of a particular application and a first order model definition. The AIM specifies a well-defined information exchange structure, organized for machine interpretation. The Conformance Testing Methodology specifies the process by which an AP will be accepted.

A major contribution to the field of product modeling has been the development of the EXPRESS language. EXPRESS was conceived as a means for specifying the structure of data, as an intermediary between an ARM and a Physical Implementation. It was conceived to support a variety of implementations, including flat file formats—either binary or ASCII—and also relational or object-oriented database formats. Originally, it was assumed that the translation from an ARM to an AIM would be done manually, and the ARM would serve to verify that the AIM was specified correctly. However, mapping tables were developed early that specified regular ways to translate the constructs of the IDEF1x and NIAM data modeling languages into an equivalent EXPRESS construct. Recently, translators or compilers that automatically generate an EXPRESS model from a NIAM or IDEF1x model have been developed. This has simplified the generation of Application Interpreted Models. Also, STEP activities have increasingly used EXPRESS—especially in its graphic format (EXPRESS-G)—for defining ARMs. Translation between EXPRESS-G and EXPRESS is widely supported.

Interpreted Resources provide a set of resource models that may be utilized within different Application Protocols. They are specified as an AIM, facilitating their assimilation into multiple APs. As ARMs are interpreted into AIMs, they are adjusted to allow integration of relevant Integrated Resources. Two levels of resources have been defined: Generic Resources, which have general use across applications, and Application Resources, which support one application or a cluster of similar applications. It has been assumed that Application Resources would be incrementally defined *post facto*, as parts of models developed in one application are found to be needed in others. More recently, this approach to Application Resources is being reconsidered.

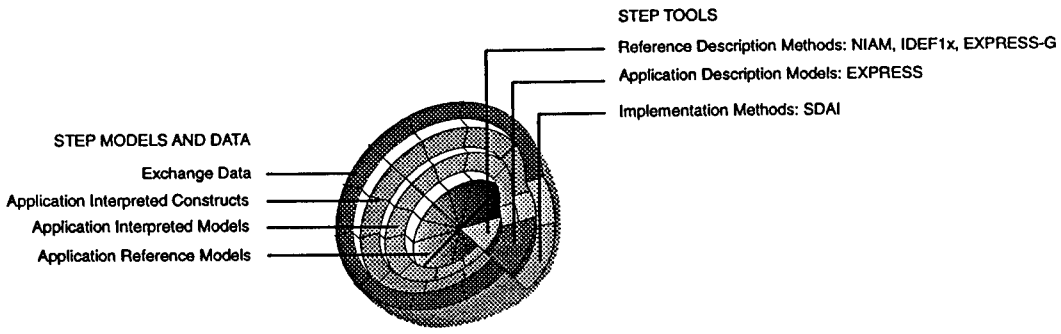

Figure 5.2: A layered representation of the ISO-STEP architecture.

Another way to visualize the STEP architecture considers STEP resources and models in terms of a layered system architecture, as shown in Figure 5.2. The base facilities are defined at the center, with the facilities that use them outside, in layers resembling an onion. The wedge in the lower right of the onion represents language resources. Thus information flows are between layers in Figure 5.2. The figure shows the three levels of the application model for generating exchange data, with reference and interpreted models being supported by Integrated Resources. The outer layers build upon the resources of the interior ones.

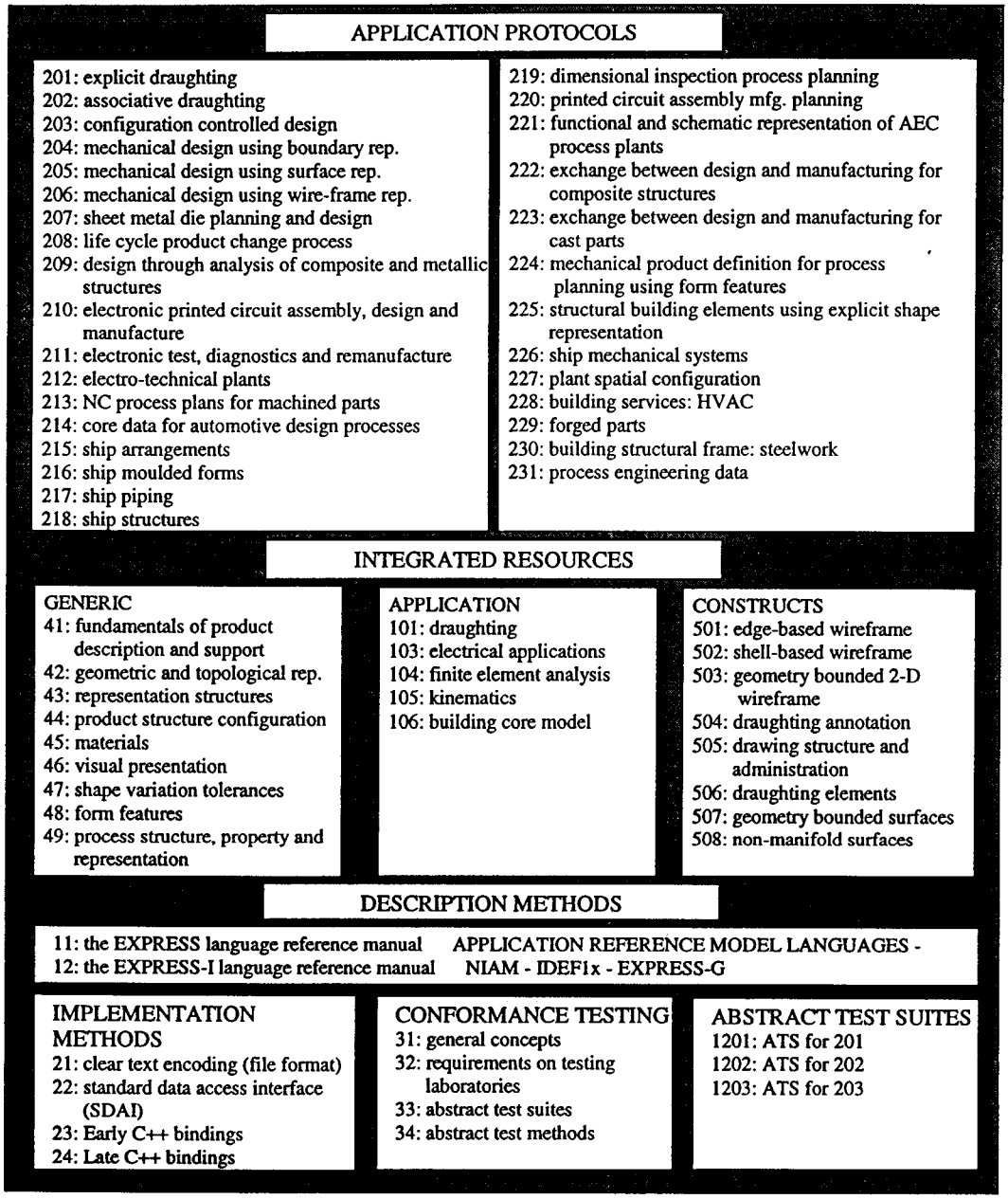

Figure 5.3: The numbers and titles of STEP Parts, as specified in early 1996.

The listing of the different STEP Part definitions that have been at least initiated, as of early 1998, is shown in Figure 5.3. Each ISO1303 series is published as a separate Part. Parts are grouped into one of the following classes: Application Protocols, Description Methods, Integrated Resources, Implementation Methods and Conformance Testing, as shown in the figure. Because the need for solutions for data interchange problems has been so great, pressures have been exerted to develop even limited solutions. As a result, draft forms of many Integrated Resources and Application Protocols have been developed in parallel, even with the Description Methods not being complete.

As can be seen, the ISO 10303 efforts are wide-ranging, and it is non-trivial to comprehend their full extent. The Part models initiated for the building industry thus far are 225 (structural building elements using explicit shapes), 228 (building services, HVAC), and 230 (building structural steel frame), and the Integrated Resource Part 106 (building construction core model). In addition, we must understand the Part models that the building-specific models use. These include Part 11 (EXPRESS), Parts 41-43, which provide the ownership, state, and geometry for building parts, and Part 45, which defines materials. For implementations, we need to understand Parts 21, 22, 23 and 24.

In the rest of this chapter, we focus on Part 11 (EXPRESS) and another frequently used ARM language, NIAM. We also will review Parts 21-24 to provide an overview of implementation tools. The next chapter, Chapter Six, surveys Parts 41, 42 and 43. Chapter Seven includes reviews of Part 230 and 228 and other work on aspect models, while Chapter Eight includes a survey of Part 106 (Building Construction Core Model) and other similar efforts.

5.3 IMPLEMENTATION CONCEPT

Most of the attention and people involved in STEP activities focus on the definition of various product models, both as Application Protocols and Integrated Resources. However, a product model is not of much use if it does not support implementation of a data exchange mechanism. That is its ultimate purpose. The purpose of a logical model, represented in EXPRESS, is to define the structure and format of data instances that correspond to that model, for exchange with another application.

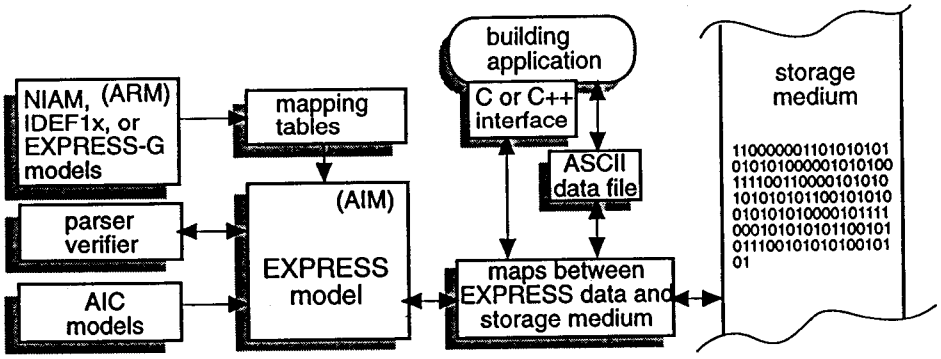

Figure 5.4: The general implementation approach used in STEP.

To gain an overall picture of the intended operation of STEP, it is useful to consider the process in three steps, which can be interpreted from Figure 5.4. The Application Protocol development process goes from the ARM (expressed in IDEF1x , NIAM or EXPRESS-G) to the AIM (represented in EXPRESS). It typically includes references to Application Integrated Constructs, for example, for defining standard geometries. The AIM is generated using mapping tables that

define the correspondence between ARM constructs and EXPRESS. The AIM is verified, in part, using a parser/verifier.

In order to write STEP conformant data into a storage medium or read it, a *mapping structure*, written in C or C++, is defined that corresponds to the Entities in the EXPRESS model. There are different methods for generating this mapping structure. It provides an in-between format between a building application and some storage medium. If the application can be extended to incorporate the mapping structure, it can copy its entities into the mapping structure's Entities. Alternatively, the application may write its entities to a file, which can be read by the mapping structure. The mapping structures have interfaces to write to various storage media, including a file and database formats. After one application writes to the storage medium, another can read it, using the same EXPRESS Entities that were used for writing. In this case, the mapping structures read the entities and either copy them to a second application's data structures, or to a file format readable by the second application. Later, in Section 5.8, we review this process in greater detail.

An important aspect of the STEP data exchange architecture is the use of mapping tables. Conceptually, a mapping table is a table of equivalencies, i.e.,

REAL in ASCII format	→	Real in IEEE double precision format
Double in EXPRESS	→	Double in C++
Cartesian point in Part 42	→	same on a file
Cartesian point on file in Part 42 format	→	Cartesian point in CAD application
Location point in CAD application	→	Location point in Part 42 format
Wall in CAD system format	→	Wall in Part 106 format

The function of the mapping table is to guide a program; when it encounters the entity on the left-hand side, it is to write out the entity on the right-hand side. Clearly, the definition of such a mapping table ranges from being simple to extremely complex. All computer languages do the first mapping whenever they read or write a real number; the second mapping should not be any harder. A Cartesian point in Part 42 is three REAL numbers, so the second and third mappings should also not be hard. But what about the last one, regarding walls? How can we define the structure of a wall and then map it to some other format? The important point to remember is that all computer models are compositions of a few primitive data types and structures for grouping them. If the primitives and structures are defined in the mapping table, all higher-level entities defined as compositions can be mapped in terms of the primitives and structures. It is generally on this basis that mapping tables are generated, as we shall see. Mappings are required in many places for data exchange: between an external application and an EXPRESS model, between EXPRESS and a text file, or between EXPRESS and a database. Mapping and mapping tables are at the heart of data exchange and will be encountered many times throughout this and later chapters.

IGES and other early, neutral file formats had the problem that the data held in the model representation also included comments, scoping rules and other information structured for human, rather than computer, recognition. It was recognized that the specification for the information model should be separate from the data itself. This is realized in the above scenario, with EXPRESS providing the comments and format instructions for the data stored. The stored data can be very compact. The data could be written to a file, to a database or other information repository. In practice, the main work has relied on files.

The following sections review three of the tools used in STEP. We examine two ARM methods: NIAM and EXPRESS-G. Both are presented in detail, but serious use should be based on the official documentation. Examples are developed in both languages, facilitating comparison. EXPRESS is also presented, with sufficient detail for beginning use.

5.4 ARM DESCRIPTION METHODS

The STEP Description Methods address several different aspects of the specification of a data exchange process. The two most widely used aspects are the Description Methods used for defining ARMs and the Description Methods used for defining AIMs. ARMs are typically defined in either the NIAM information modeling language or EXPRESS-G. Another language used is IDEF1x. AIMs are uniformly defined in the EXPRESS schema definition language. (EXPRESS-G is a graphical version of EXPRESS, allowing translation from EXPRESS-G to EXPRESS to be almost transparent.) In this section, we provide an overview description of NIAM. Since the other language for defining Application Reference Models, EXPRESS-G, is a graphical implementation of EXPRESS, it will be reviewed after EXPRESS, which is discussed in the next section.

The development of an ARM begins with the definition of the Application Protocol's scope and requirements. What functional requirements should the AP fulfill? The STEP guidelines recommend the following be used to develop an ARM:

- the classes of external applications that the AP is to support
- example objects to be represented
- usage scenarios for the AP
- a definition of the scope of the AP, regarding both what is inside and what is outside the scope of the AP

These requirements are then defined more strongly as *Units of Functionality (UOF).* UOF is a STEP term referring to the general entities, attributes and relations that convey the concepts within an AP. A critical membership criterion is applied: if any component of the UOF is removed, the concepts should be rendered incomplete or ambiguous. The UOF is usually defined in words; after agreement on it, the UOF is defined more rigorously, using an ARM Description Method. The purpose of ARM Description Methods is to specify at a detailed level the requirements for an Application Protocol or for a shared Integrated Resource. Thus, an ARM definition is the permanent, recorded basis for defining any later model.

5.4.1 NIAM

In 1977, Dr. G. M. Nijssen of CDC Europe developed one of the most widely used data modeling descriptions applied for the design of databases, the Nijssen's Information Analysis Method (NIAM). Dr. Nijssen developed NIAM as a database design methodology, supporting information exchange between a user and a computer, using elementary sentences. He had in mind relational databases and conceptualized NIAM as a means to translate written or verbal information into a database schema.

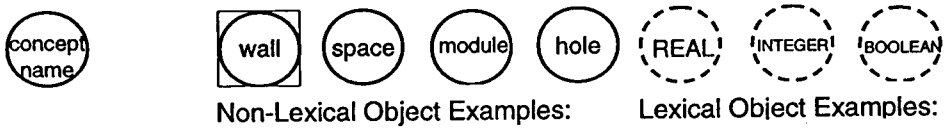

Figure 5.5: Objects in NIAM are of two types: Lexical Objects (LOTS) and Non-Lexical Objects (NOLOTS).

NIAM relies on an information-processing model of functional transformation—that is, it uses functions to transform information from one form to another. It provides a small set of constructs, which are the building blocks with which users can define higher-level constructs. There are two main primitives. One is the *object*, which is of two types: the *lexical object type* (LOT), and the *nonlexical object type* (NOLOT). The object in NIAM does not have any relation with object-oriented (OO) systems. A NOLOT corresponds closely to the concept of entity; it may depict

conceptual classes of things or large aggregations of attributes. LOTS are those objects that are names or values of NOLOTS and have a type and coding rules. They are used for identifiers or attributes. LOTS are dependent upon the NOLOT they describe and are deleted if their NOLOT is deleted. Their graphical representation is shown in Figure 5.5. An Object with an enclosing box is a non-terminal, indicating that the details of the Object are defined elsewhere. Object is a class that has instances (called Entity in EXPRESS). Thus in Figure 5.5, there may be a variable number of `wall`, `space`, `module` and `hole` members. There may be a variable number of REAL, INTEGER and BOOLEAN LOTS.

The other main NIAM primitive is the *Role*, which corresponds to a relationship between Objects. Roles are usually binary and bi-directional—that is, they usually exist between two Objects and define a reciprocal relation. Occasionally, Roles have more than two Objects. The graphical notation for Roles is shown in Figure 5.6. Each Role has a box for each Object associated with it, with a name or attribute for the Object's part in the Role.

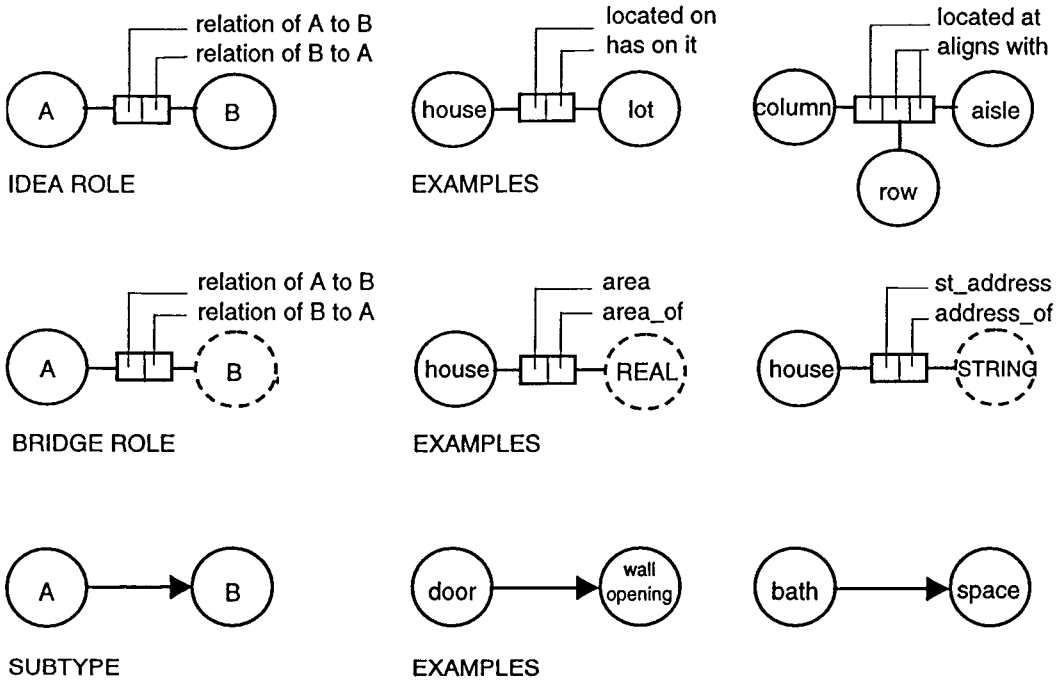

Figure 5.6: Roles within NIAM are usually binary. Roles are of two types: idea and bridge. A special type of Role is subtype, where the NOLOTs and Roles of one object are also those of another object.

Roles are of different types. Idea Roles are between NOLOTS—for example, between lot and house or between column, row and aisle. Bridge Roles are between a NOLOT and the LOTS that describe it. The box descriptors are the attribute names for the LOT. Subtype is an inheritance relation between two NOLOTS. These three Roles are the base structures within NIAM. The semantic richness of NIAM, however, arises from its many constraints, which are rules restricting the form of Role allowed between LOTs, NOLOTs, idea types, bridge types and subtypes.

One set of major constraints deals with *Uniqueness*. Some examples of Uniqueness constraints are shown in Figure 5.7. The bar on one side of a Role box indicates that that side has zero or one member in the Role and that no duplicates are allowed. Uniqueness allows one Object to identify the other. A continuous bar indicates that both Objects may be duplicated but the Role cannot. No

bar indicates that the Role can be duplicated as well as the Objects. In the examples shown, a house may be on zero or one lot, but a lot may have any number of houses. A wheel is on at most one car, but a car may have multiple wheels. On the other hand, a door has only one threshold, and a threshold has only one door. Similarly, an auto has one engine, and an engine has one auto. A room is bounded by multiple walls and a wall bounds multiple rooms. Last, a room may be accessible by zero, one or more corridors, and a corridor may provide access to zero, one or more rooms. Thus a one-sided bar indicates a one-to-many relation, bars on both sides (not continuous) indicate a one-to-one relation and a continuous bar indicates many-to-many.

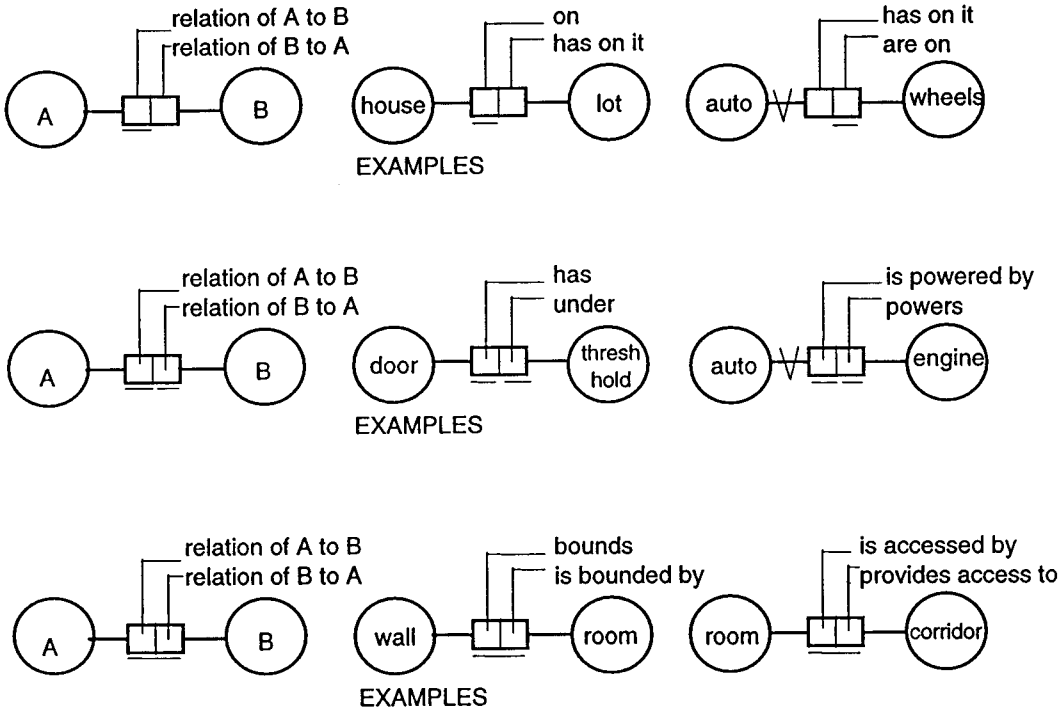

Figure 5.7: The uniqueness constraints delimit the arity of a Role and also the uniqueness of values. Uniqueness may be one-to-many, one-to-one, or many-to-many. The Total constraint is shown with a V intersecting the Role line.

Uniqueness is a fundamental relation in NIAM because of the major impact it has on the design of relational database schemas. Uniqueness provides a general definition of the arity of a Role. Sometime a specific number of Objects in a Role are required. Optionally, arity can be defined by noting the lower and upper bounds on the number of Objects that may participate in a Role. Examples are shown in Figure 5.8. This may be meaningful for Roles that are not one-to-one.

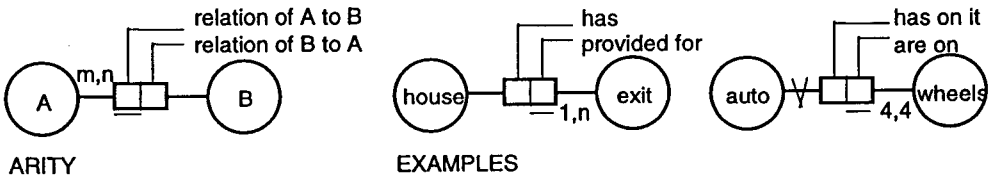

Figure 5.8: The arity constraints delimit the bounds on the number of objects that may be members of a Role.

Another important constraint is the Total constraint. It enforces the condition that at least one Object instance in a relation must exist. If it is deleted, the other Object in the relation is also

deleted. Thus it defines dependency. In the examples in Figure 5.7, the wheels are dependent upon the auto and below it, the engine is dependent upon the auto. The total constraint is denoted by a V intersecting the Role line. In the two examples, if the auto is deleted, so are all its wheels and so is its engine.

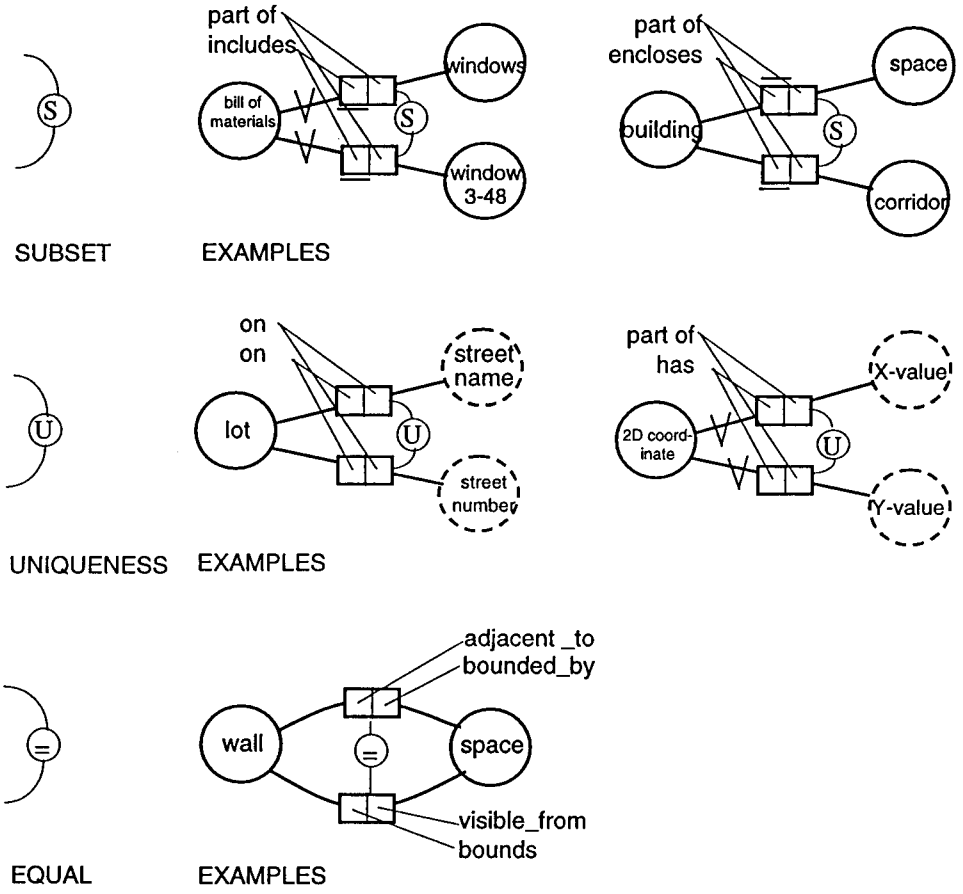

Figure 5.9: Constraints that apply to Roles are the subset, uniqueness and equality constraints.

Other constraints apply to multiple Roles. The subset, uniqueness and equality constraints are diagrammed in Figure 5.9. The subset constraint may apply to LOTs or NOLOTs. It specifies that an instance value for one attribute must also be an instance value of another attribute. In the examples of Figure 5.9, all specific window models are members of the set of all windows, and all corridors are members of the set of all spaces. The uniqueness constraint identifies a set of LOTs that uniquely define a NOLOT. The equality constraint requires that there are equivalent sets of Objects in the related Roles. In the example, one relation defines adjacency and the other defines visibility; both have the same membership.

Another set of constraints—inheritance constraints—applies to Subtypes. The mutual exclusion constraint imposes that no member of one subtype may also be a member of another subtype. As shown in Figure 5.10, an opening may be a door or a window, but no instance may be a member of both NOLOTS. (Sliding glass doors cannot be defined as members of both, for example.) Also, an instance of vertical circulation cannot be both a member of stair and elevator. Another Subtype constraint is Total, which means that there are no members of the supertype that are not members

of the subtypes. For example, all pass-thrus are either flexible pass-thrus or rigid pass-thrus. Similarly, all members of vertical circulation are either stairs or elevators (dumbwaiters are excluded).

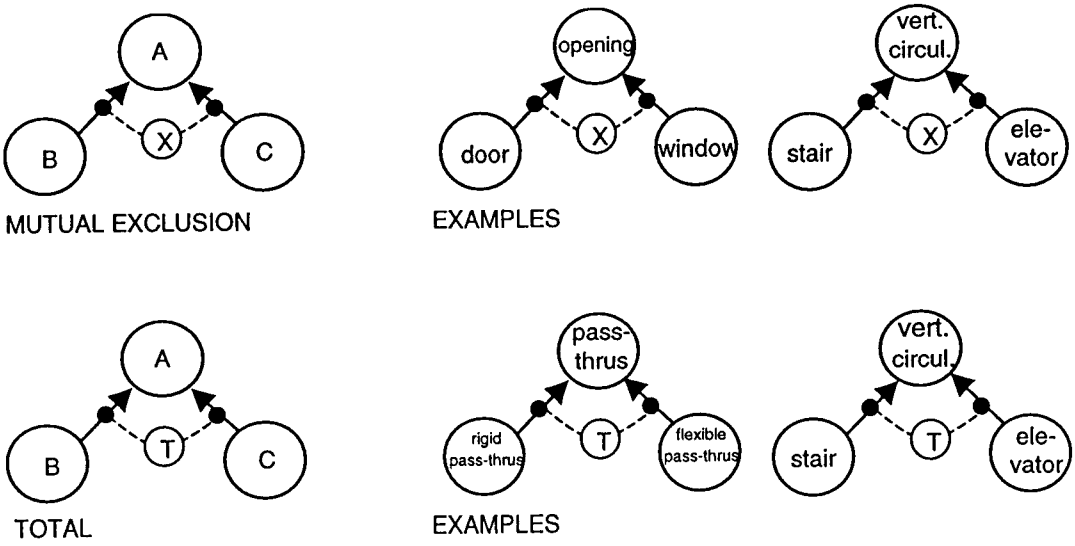

Figure 5.10: Mutual exclusion and total constraints apply to subtypes.

Supposedly, the conceptual basis of NIAM is natural language. It was conceived, however, to generate as output a relational database schema and is supported by various tools sold by Control Data Corporation and others. Its semantics were defined so as to have an implementation in relational databases. A variety of special constraint implementations have been developed for relational databases that correspond to the NIAM constraints defined above.

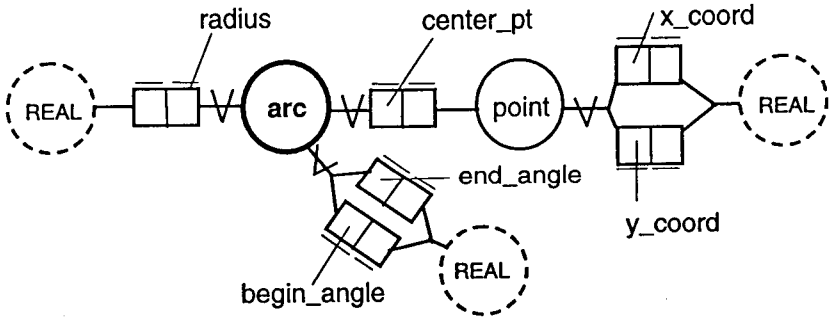

Figure 5.11: An arc modeled in NIAM.

5.4.2 NIAM EXAMPLES

In order to demonstrate the NIAM information-modeling language for product modeling, we use it to model the three examples introduced in Chapter Three: the arc, the bounded planar surface and the multi-view wall.

The arc is shown in Figure 5.11, defined as a point, two angles and a radius. All information is dependent upon the arc Object; if it is deleted, the other parts are also. All entities are unique, without duplicates. There is exactly one of each.

The bounded plane is shown in Figure 5.12, defined as a surface and a bounding polygon. In both cases, the definitions are simple: the surface is one of several subtypes, one of which is a plane. A surface can be of only one of these types. The plane is defined as a single point and single vector indicating its slope. The polygon is a sequence of points, where a point is defined as 0 to 3 coordinates. Again, all parts are dependent upon the bounded polygon; if it is deleted, the parts are also. All parts of the surface are unique to the surface and one-to-one. There may be multiple bounding polygons, however.

Aspects of the wall are defined separately, then combined later. The first part of the wall model defines the aggregate description of the overall wall and its top-level components (see Figure 5.13). The aggregate wall is defined here in terms of its geometry and thermal properties (other properties could easily be added). It is bounded by a number of boundary entities, of which wall is one subtype. The wall geometry has three views: plan, elevation, and BRep model. Each is defined by its component geometric entities. The geometry level carries a location for all the component geometric descriptions. The symbol has a many-to-one relation with geometry, while the location, solid and p-line have one-to-one relations. Multiple p-lines are defined supporting both elevation and plan. The thermal property has a single value, the U-value. All properties are dependent upon the wall, except pass-thru, which is defined independently of the wall and hence will not be deleted if the wall is.

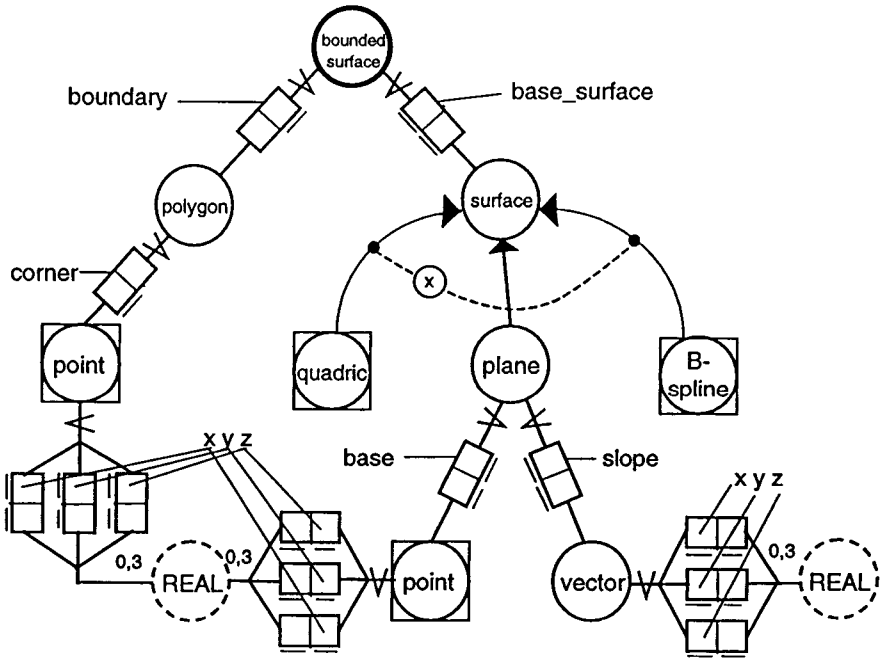

Figure 5.12: The bounded plane example, defined in NIAM.

As its parts, the wall has openings, pass-thrus, and segments. Segments and openings are dependent upon the wall, and have a many-to-one relation. Pass-thrus have a many-to-many relation and are not dependent upon a wall. Segments, openings and pass-thrus are all references to other definitions.

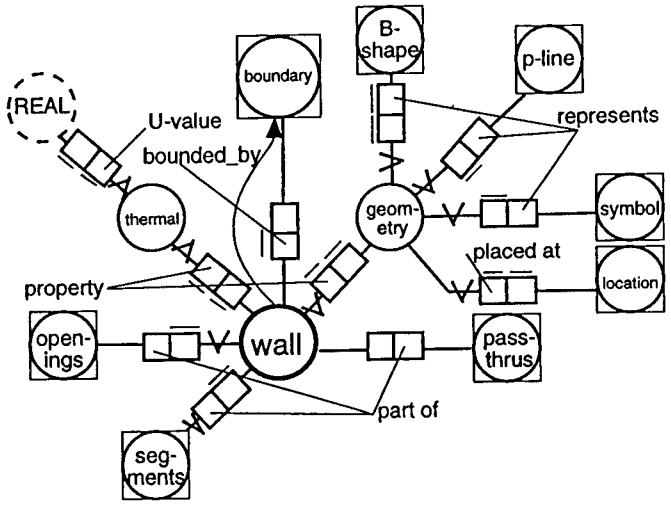

Figure 5.13: The top-level definition of the core wall, defined in NIAM.

The definition for openings is shown in Figure 5.14. Every opening has a thermal U-value attribute and a geometry Object carrying a location, an area, and a polyline outlining the opening. Filled openings are specialized from openings to carry the geometric definition of the filler. Two views are offered: a symbol for plan and elevation and a solid for 3D modeling. Windows and doors are specializations of filled openings.

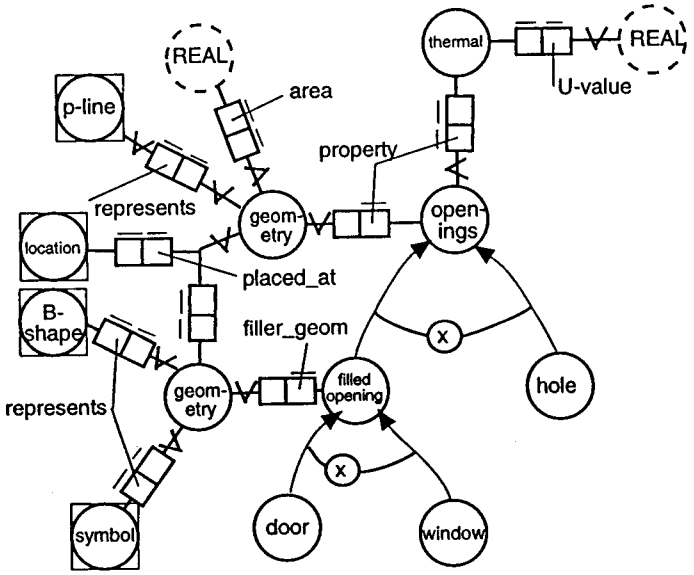

Figure 5.14: The NIAM definition of openings within a core wall.

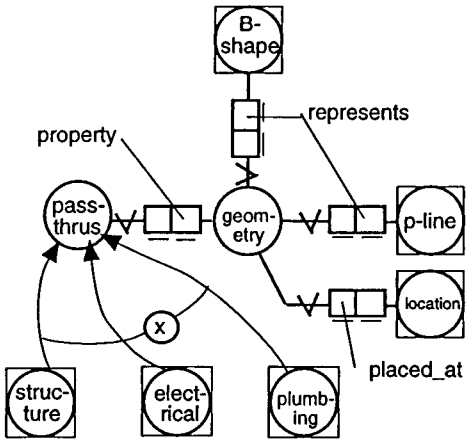

Figure 5.15: The NIAM definition of pass-thrus for the core wall definition.

A pass-thru is a generalization of any Object that passes through the wall using it as a chase (Figure 5.15). Pass-thrus are assumed to have one or two views: a centerline for rough layout (p-line) and possibly a solid model for detail layout. Geometry carries its location. Specializations of pass-thrus include structural, electrical and plumbing entities.

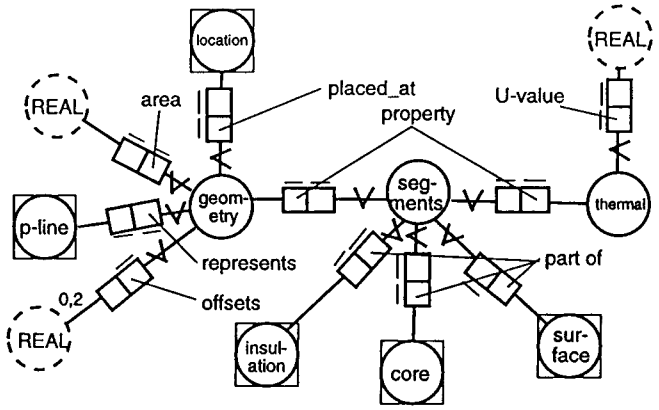

Figure 5.16: NIAM definition of a wall segment, its geometry and parts.

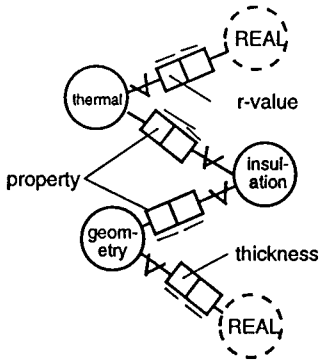

Figure 5.17: NIAM figure of wall insulation, as part of segment.

The major construct defining the solid part of a wall is the segment (Figure 5.16). A segment is a region of a wall with a consistently layered cross-section. At this level, a segment geometry specifies its location and carries a polyline defining its extent in elevation and the corresponding surface area. The segment geometry also has two offsets to each side of the wall, defining the thickness for this segment. It also has a U-value as the partial result from thermal analysis. The segment is further broken down into core, insulation and surface. All properties are dependent upon the segment, i.e., there is a total constraint. Most Objects have arity one. The coordinates are constrained to have up to two values.

Insulation has two properties: its resistance, or r-value, and its thickness, which often determines its r-value (see Figure 5.17). All properties hold one-to-one relations and depend upon the insulation Object.

The core of a wall segment defines its inner structure, which is assumed to provide its structural rigidity (see Figure 5.18). It is defined geometrically by a location, two offsets from a centerline, and a solid model (for exact shape definition). The core also has a thermal r-value. The core has a variety of specializations, some of which are included.

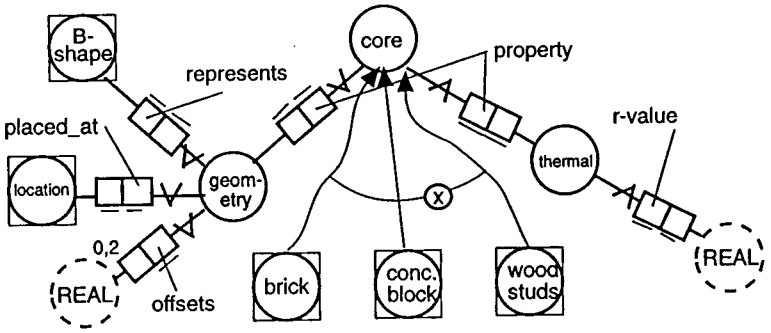

Figure5.18: NIAM definition of core of wall segment.

Surfaces are layered on both sides of the core to make up the finished segment (see Figure 5.19). Multiple surfaces may be laid on top of one another. The geometry of a surface is defined according to the side it is on, its order of placement and its thickness. An attribute r-value is included.

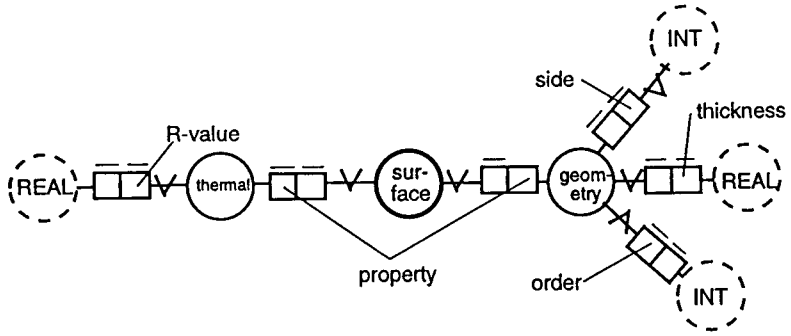

Figure 5.19: NIAM definition of the surface of a segment.

All of these aspects are combined into the overall NIAM core wall model, shown in Figure 5.20. It suggests the necessary complexity of a wall definition needed to support a range of applications. Some examples of core construction materials are offered as subtypes in the model.

5.4.3 SUMMARY

NIAM is an evolving language with alternative notations. The original documents present it as a conceptual-modeling language, meant to capture the semantics of a situation, independent of any database implementation. Later it began to acquire properties needed for automatic schema definition of relational database models. This evolution is evident from early publications describing the language to the most recent ones. We have presented a version that is an extension of the earlier versions, with an expanded set of attributes, which seems most commonly used within the product-modeling community. While this section presents most of the constraints developed or proposed for NIAM, new ones are added every few years.

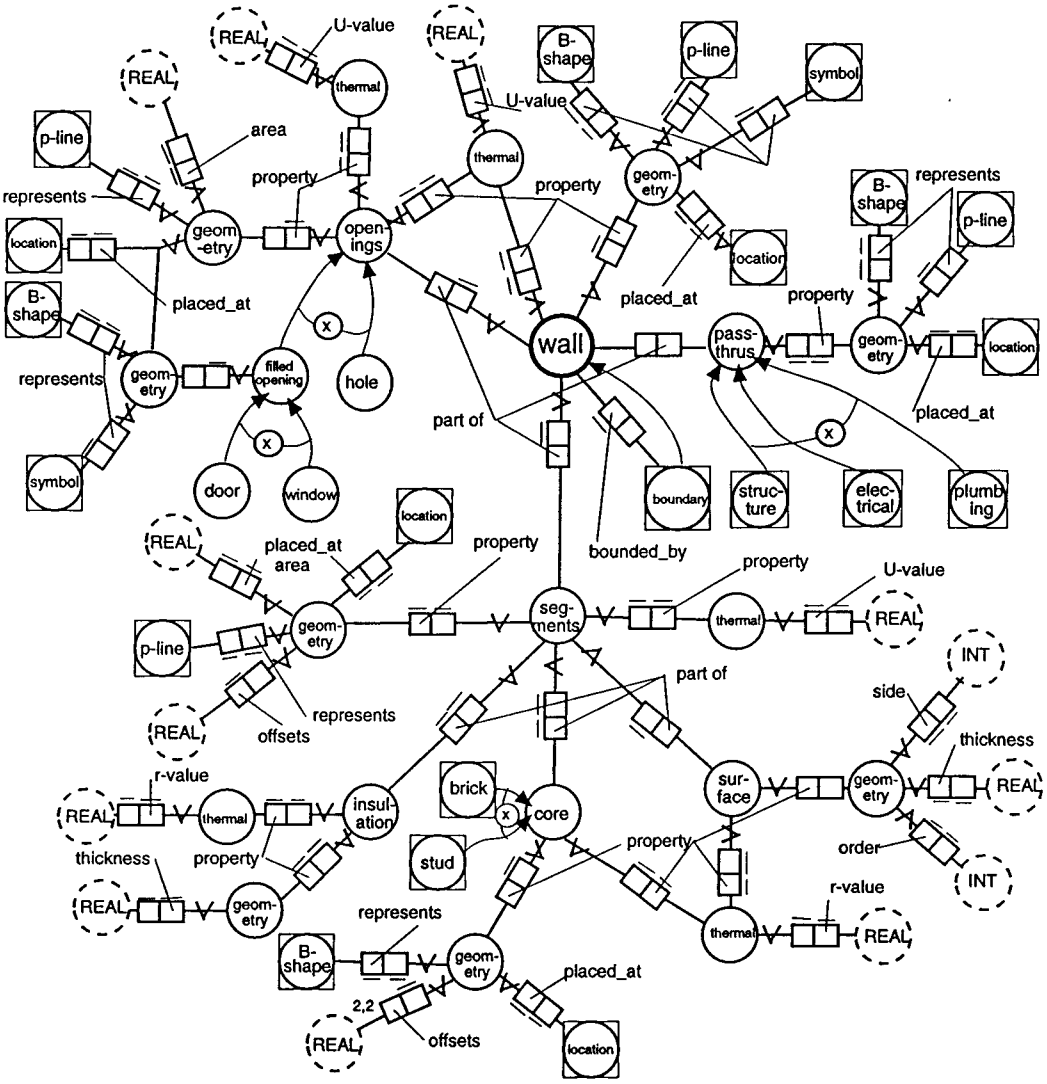

Figure 5.20: The various parts of the NIAM wall model combined.

NIAM's entity structure is extremely simple, defined in terms of Objects and Roles. It is elaborated by many kinds of constraints, some of which have subtle definitions. The texts written by the developers of NIAM emphasize the conceptual and intuitive aspects of modeling, but these seem ambiguous if there are not also clear operational distinctions. For data modeling, the operational distinctions are the specific forms of the data model required. A strength of NIAM is that it makes no distinction between a relation (defined as a Role) and an attribute. This distinction is imposed in some other models and, as such, is sometimes unclear. Also, NIAM provides a rich set of constraints or validation rules for which implementations have been defined in relational databases.

Yet for all this richness, several limitations are apparent for NIAM's use in product modeling. It has no means to represent several important constructs that are also missing from the relational model. For example, the rule that the order and side of a surface must be unique for a segment, and that the order must be a complete numbering, cannot be easily expressed. Composition rules that specify what is allowed in a particular composition cannot be specified. Also the consistency of

geometrical relations, or the values carried at various levels of aggregation for thermal performance, cannot be expressed. As a graphical language, its use has been to define new schemas from scratch, not to revise existing schemas. Thus, it does not support model extension or evolution.

Though having a few shortcomings, NIAM has a significant audience and continues to be used to develop ARMs for various new application protocols within the STEP community.

5.5 EXPRESS - THE AIM DESCRIPTION METHOD

EXPRESS is the language developed within the ISO-STEP community for representing application interpreted models. It came into existence in a US Air Force project named Product Definition Data Interface and contracted to McDonnell Douglas. EXPRESS was developed by Douglas Schenck and has been under development since about 1986, when it received its current name. The purpose of EXPRESS is to represent a product model in an implementation independent manner, as interpreted from an Application Reference Model (ARM). That is, it is a representation of the knowledge to be embedded in a product model, interpreted to take advantage of shared Integrated Resources (reviewed in Chapter Six) and rationalized with other related application protocols.

EXPRESS's syntax is similar to that found in modern programming languages. It is used in the way one would define the data structures in a programming language. Its effect is also similar, in that it defines how instances of defined objects will be organized for use. It is different from a programming language in a few important aspects, however, and these differences must be kept in mind when examining or using it. This section presents its functionality along with its syntax. The following sections provide a working overview of EXPRESS Version 1.0.

5.5.1 SCHEMAS

The unit of definition in EXPRESS is a *schema*. A schema defines the *universe of discourse* (UoD) in which declared objects are given mutually dependent meanings and purposes. An EXPRESS schema is the normal representation of both AIMs and Application Interpreted Constructs (AICs). As defined in the EXPRESS manual, an EXPRESS information model consists of schemas that include *definitions of things* (entities, types, functions, procedures), rules defining *relationships between things*, and *rules on relationships*. EXPRESS includes a full, procedural language syntax for specifying rules. The rules defined in EXPRESS cannot be executed, however, but serve as guides for the implementation of translators. (Proposals exist to change this.)

EXPRESS is a block-structured language, like Pascal or C. All types have scopes, which are identified by the block in which they are defined. A block begins with the declaration of an Entity, Function, Procedure Rule or schema and ends at the end of Entity, Function, Procedure Rule or schema. When an identifier in one block is redefined in an inner block, the inner block declaration overrides the outer one, for the extent of the inner block. Any declarations outside a schema are within a global block referred to as the UoD.

An information model may be defined by more than one schema. There are special mechanisms to make cross references across schemas, but they require a different syntax in relation to references within the same schema. Thus, one schema can be linked to other schemas using the USE and REFERENCE specifications.

EXPRESS code may include comments and other non-interpreted text. These are bracketed with (* and *). Comments may also be appended to a line with --.

```
--Here is a comment:
(* Here is another comment. *)
```

5.5.2 IDENTIFIERS

The general convention that is followed is to define identifiers in lower case, and to capitalize all reserved words. Identifiers must begin with a character and may include other characters, digits and the underscore character. The characters for blankspace ( ), tab, arithmetic operations, period (.) and comma (,) are not allowed in identifiers.

5.5.3 BASIC TYPES

EXPRESS provides a set of basic types that are predefined and available to use in the definition of higher level types.

The Basic Types are

```
NUMBER, REAL, INTEGER, STRING, LOGICAL, BOOLEAN and
      BINARY.
```

INTEGER is an unconstrained whole number. REAL is an unconstrained rational, irrational or scientific number. Scientific numbers may be constrained to a specified precision. NUMBER is a supertype of INTEGER and REAL. STRING is a quoted list of characters, bounded by single quotes. LOGICAL types have the values (TRUE, FALSE, UNKNOWN) while BOOLEAN only has (TRUE, FALSE). BINARY is a vector of binary values, with user-defined encoding.

Both STRING and BINARY are variable length vectors. They may have a specified length. When a length is defined, up to that many may be assigned. When the length is specified and appended by FIXED, then all assignments must be of exactly that length. But, if it is not specified, the length is variable. For example

```
s1: STRING;        (* variable length string *)
s2: STRING(10);    (* variable length up to 10 characters*)
b1: BINARY(10) FIXED;          (* exactly 10 bit flags *)
```

EXPRESS also has an enumeration type, where the possible values are defined explicitly. For example

```
TYPE compass_direction =
  ENUMERATION OF (south, north, east, west);
END_TYPE;
```

5.5.4 CONSTRUCTORS

Variables and other structures can be aggregated into larger groupings, using *constructors*. Constructors group things of the same type; in EXPRESS they include the `array`, `bag`, `list` and `set`.

ARRAY is used to define an ordered list of elements of fixed size. Arrays may be concatenated, e.g.,

```
matrix : ARRAY[1:4] OF ARRAY[1:4] OF REAL;
```

In ARRAY, both the lower and upper bound must be defined (where lower_bound < upper_bound). There will be (upper_bound - lower_bound + 1) items in the array. Items are accessed by the array name and a subscript, e.g., `matrix [3][1]`.

BAG is an unordered collection of like elements. Duplicates are allowed. Its lower bound and upper bound may or may not be specified. If the lower bound is specified, there must be at least the lower bound of things assigned to the bag. If the upper bound is assigned, it indicates the maximum number of members it can hold. If the bounds are not defined, it is assumed that the bounds are [0:?]. As an example,

```
bag_of_points : BAG[1:?] OF point;
```

means that there must be at least one point in bag_of_points.

LIST is an ordered collection of like elements, similar to the ARRAY, but LIST may have a variable length. Thus:

```
list_of_points : LIST[0:?] OF point;
```

means that there may be any number of points in the LIST, including zero. The LIST may grow or shrink but remains ordered, with the subscripts as assigned.

SET is an unordered collection of like elements. Duplicates are not allowed. SETs may be of fixed or variable size. An example might be

```
set_of_names : SET OF [1:100] OF name;
```

set_of_names must have a membership of at least 1 and not more than 100 members. If either or both the lower and upper bounds are left undefined, the default is that there may be zero or any number of members in the SET.

A generalization of all the constructor types is the AGGREGATE. AGGREGATE may be used in any declaration where any of the constructors may be utilized.

In general, access to items in any aggregate (ARRAY, BAG, LIST, SET) is by using subscripts ranging from the lower to the upper bounds.

Basic Types may be used to specify higher level user-defined Types, e.g.,

```
TYPE area = REAL;
END_TYPE;

TYPE name = STRING;
END_TYPE;
```

A type definition must end with END_TYPE.

Another type definition allows specification of a type that is a selection from among a set of types. EXPRESS calls this a SELECT TYPE. In other languages, such a construct is sometimes called a union type. A SELECT TYPE is also a generalization of other types. Some examples follow:

```
TYPE NUMBER = SELECT(REAL,INTEGER);
END_TYPE;

TYPE connection = SELECT(nail,screw,bolt,glue);
END_TYPE;
```

Constructors extend the means to specify Defined Types from Basic Types, and to specify other Defined Types from Basic or Defined Types. A limitation is that all elements of a constructor are of the same type.

```
TYPE column_row = LIST[1:?] OF line;
END_TYPE;

TYPE column_aisle = LIST[1:?] OF line;
END_TYPE;

TYPE color_value = INTEGER;
END_TYPE;

TYPE rgb_color = ARRAY[1:3] OF color_value;
END_TYPE;
```

The first two types listed above are defined for row and aisle. By starting with one, they are required to have at least one member in the list (not that the subscripts start at 1). If no subscript is defined for a constructor type, the default is `[0:?]`. Constructor types are only defined to provide high-level types to be used in attributes. That is, they are not Entities and cannot be instantiated.

5.5.5 ENTITIES

The general object type, from which instances of objects are made, is called the Entity. It supports definition of a wide range of complex elements, which we will consider incrementally below.

The basic Entity definition has the format:

```
ENTITY point;
  x,y,z : REAL;
END_ENTITY;
```

This point may have instances that may be independent of any line or arc. Instances of `point` may be defined, while a type may only be used to define an Entity or another type.

Entities can be inherited or subtyped into other Entities. For example,

```
ENTITY homogeneous_point
  SUBTYPE OF (point);
  w : REAL;
END_ENTITY;
```

An equivalent declaration would be for `point` to be declared as a `SUPERTYPE` of `point1`. An Entity may define both `SUBTYPE` and `SUPERTYPE` relations with other types. It may be a `SUBTYPE` to zero, one, or more than one other types. Similarly, an Entity may be a `SUPERTYPE` to any number of subtypes. Either the `SUPERTYPE` or the `SUBTYPE` may define a relation between the two Entity types. One or both Entity types may carry the relation declaration. However, all `SUPERTYPE` and `SUBTYPE` declarations, taken together, must be consistent with a directed acyclic graph. That is, an Entity cannot both a `SUBTYPE` and a `SUPERTYPE` of the same Entity, at any level of the relationship. There can be no cycles in the Entity subtype graph.

A `SUBTYPE` inherits all attributes of the `SUPERTYPE`, including `DERIVED` and `INVERSE` attributes. Domain constraints defined as `WHERE` clauses (discussed in the next section) are also inherited. Overwriting of rules, allowed in some languages, is not allowed in EXPRESS. This

restriction addresses the consistency of polymorphic types (discussed in Chapter Four) in a strong manner.

5.5.6 ATTRIBUTES

Attributes in EXPRESS are of three general kinds:

Explicit: the values will be provided directly

Derived: the values can be calculated from other attributes

Inverse: captures the relationship between the Entity being declared and a named attribute

The attributes have a representation, which might be a simple data type (such as integer) or another Entity type. A relationship is established between the attribute being defined and the types or Entities that define it. In the default case, the schema instance is only correct if a value exists for the attribute, that is, if the relation is mandatory. If an empty value is allowed, an attribute is declared as `OPTIONAL`.

5.5.6.1 Explicit Attributes

Explicit attributes are the information units used to characterize the properties of some Entity. They may be classified as

Simple types: These cannot be further subdivided into elements; they are `NUMBER`, `REAL`, `INTEGER`, `STRING`, `BOOLEAN`, `LOGICAL` and `BINARY`

Aggregate types: These are groupings of the same type using one of the constructors

Entity type: These are object types declared by Entity declarations. Like other attributes, using an Entity as an attribute's data type establishes a relationship between two Entities

Qualified attributes are used to define a pathname from the current Entity to an attribute within an inherited Entity.

Inverse attributes capture the relationship between the Entity being declared and any named attribute that uses it as an attribute value. That is, inverse attributes define a backpointer referencing an Entity that uses the current one as an explicit attribute. Inverse attributes may be constructor types with cardinality or other constraints.

5.5.6.2 Derived Attributes

In addition to explicit attributes that are given assigned values, EXPRESS also supports derived attributes, which are not loaded as data, but derived from other values carried within an Entity. Derived attributes are identified by `DERIVE`:

```
ENTITY circle3d;
  center : point;
  radius : REAL;
  axis : vector;
DERIVE
  area : REAL := pi * radius ** 2;
END_ENTITY;
```

Thus the attribute area can be accessed just like other attributes, but is computed at the time it is accessed. The scope of attributes that can be accessed within a `DERIVE` expression is the Entity in which the `DERIVE` expression is located.

EXPRESS also provides a `UNIQUE` clause that can be applied to any attribute. It specifies that each instance of the Entity must have a unique value of that attribute that is not duplicated anywhere within the `SCHEMA`.

5.5.6.3 Inverse attributes

Often, an attribute defines a relation between the Entity carried by the attribute and some other Entity. For example,

```
ENTITY line;
   point_ref : ARRAY[1:2] OF point1;
END_ENTITY;
```

defines a relation between a line instance and two point instances. Sometimes we may require that the relationship is two-way. That is, a relation also exists from the point to the line it bounds. This may be defined as

```
ENTITY point1;
  x,y,z : REAL;
  line_ref : SET OF line;
END_ENTITY;
```

where only points that are part of a line may be of type `point1`. But now we have two relations that are related to each other. There are supposed to be two matching attributes; if `line` instance `a` has a `point_ref` attribute referencing point instance `b`, then point instance `b` should have a `line_ref` attribute referencing line `a`. Multiple lines may refer to a `point1` instance. Any updates to a line or point instance would have to maintain this relation between the attributes.

The `INVERSE` attribute imposes a symmetry rule between an existing relation and itself. It automatically creates and maintains the relation between the attributes, always defining the inverse automatically from the relation. For the `point1` Entity above, it would be defined as follows:

```
ENTITY point1;
  x,y,z : REAL;
INVERSE
  line_ref : SET OF line FOR point_ref;
END_ENTITY;
```

The `line_ref` attribute of `point1` is the inverse of the `point_ref` attribute in Entity `line`. Any time that `point_ref` is updated, `line_ref` in the affected `point1` will be also updated automatically.

5.5.7 RULES

EXPRESS supports the definition of a variety of rules that can implement many kinds of NIAM constraints, as well as other semantic conditions of importance to product modeling. It provides a variety of structures for embedding these rules. The derived attributes presented earlier describe one such structure. Domain rules describe another. They allow definition of restrictions on allowed values or combination of values in the attributes of an Entity. Domain rules are specified using a `WHERE` clause within an Entity specification.

```
ENTITY vector;
  a, b, c :         REAL;
WHERE
  length1 : a**2 + b**2 + c**2 = 1.0;
END_ENTITY;
```

The domain rule `length1` is an integrity constraint of type `LOGICAL` (all `WHERE` clauses are of type `LOGICAL`). When accessed, it evaluates the expression and returns one of the values: `TRUE`, `FALSE` or `UNKNOWN`. `UNKNOWN` is used when some attributes are missing. If `length1` is not `TRUE`, then an instance of `vector` does not conform to this specification.

In order to define complex rules, EXPRESS incorporates a fairly complete set of system functions. These include

ABS	-	absolute value	LOG10	-	base 10 log of value
ACOS	-	arc cosine	LOG2	-	base 2 log of number
ASIN	-	arc sine	LOINDEX	-	actual lower bound of values
ATAN	-	arc tangent	NVL	-	converts NULL to given value
BLENGTH	-	number of bits in a binary	ODD	-	TRUE if value is odd
COS	-	cosine	ROLESOF	-	returns set of strings of references to value
EXISTS	-	TRUE if value exists	SIN	-	sine
EXP	-	*e* to power of value	SIZEOF	-	no elements in aggregation
FORMAT	-	string of a number	SQRT	-	square root of value
HIBOUND	-	declared upper bound of constructor	TAN	-	tangent of value
HIINDEX	-	actual number of values	TYPEOF	-	set of strings of all types of a value
LENGTH	-	number characters in string	USEDIN	-	set of strings used in given Role
LOBOUND	-	declared lower bound of constructor	VALUES	-	numerical value of string
LIKE	-	matches substring of string			
LOG	-	natural log of number			

As can be seen, many of these are trigonometric and algebraic functions. The bounds checking, SIZEOF, USEDIN and other functions are provided to facilitate querying a product model. These can be used with the computational part of EXPRESS (Section 5.5.9) to define more complex and domain-specific functions for use in rules within a schema. General functions can be declared and used in defining DERIVE clauses or WHERE rules. An example is shown below. It computes the distance between two 3D points, where the points are as they were defined earlier.

```
FUNCTION distance (p1,p2 : point) : real;
(* computes the Euclidean distance between two points *)
  LOCAL  dist : REAL;
  dist := SQRT((p2.x - p1.x)**2 + (p2.y - p1.y)**2 +
                                  (p2.z - p1.z)**2);
  RETURN (dist);
END_FUNCTION;
```

Access to a particular value is gained through a path accessed by a *pathname*. Pathnames are a sequence of attribute names connected by periods (.) traversing through an instance model. In the above example, p1 and p2 are the local names of parameters of type point within the function. Point has attributes x, y and z which are sequentially accessed from p1 and p2.

In writing a function to be embedded in a WHERE clause, there is no parameter that refers to the current instance. SELF is a primitive function that serves this purpose. It may be used in Entity and Type declaration or instance initializations. For example, the variables available in homogeneous_point above are

```
SELF\point.x;
SELF\point.y;
SELF\point.z;
w;
```

In the above pathnames, SELF refers to the current Entity. The \point refers to an Entity inherited into the current Entity. These pathnames are used to access these attributes from within homogeneous_point.

A well-known issue in object-oriented systems is the resolution of duplicate names inherited from multiple disjoint SUPERTYPEs. In EXPRESS, these conflicts are resolved by defining the pathname from which the attribute name came. Thus, if an Entity `window` is a SUBTYPE of both a `wall_opening` Entity with a `location` attribute, and a `light_source` Entity with a `location` attribute, each would be accessed using the following:

```
SELF\light_source.location
SELF\wall_opening.location
```

5.5.8 RELATIONS

Like object-oriented languages, EXPRESS does not have a structure for defining relations. Like most languages, they must be defined using attributes. On the other hand, the semantics defined in NIAM and other data modeling languages need to have a well-defined implementation.

5.5.8.1 Defining the Arity of Relations

Because of the prevalence of arity constraints in NIAM and other data modeling languages, EXPRESS had to have effective ways of representing them. While most languages allow a user to define a relation in one direction (either one-to-one or one-to-many), few allow many-to-many. That is, in the definition of a polygon made up of a list of edges, it is straightforward to define relations from polygon to edges. Using the `line` Entity defined earlier,

```
ENTITY polygon;
  edges : LIST [3 : ?] of line;
WHERE
  2_connected : (* check that the line's endpoints
                        are all two-connected *)
END_ENTITY;
```

Here, the polygon may point to three or more `lines`, indicating a three-to-many relation. This shows how a constraint on the arity can be defined in EXPRESS. The earlier `line` definition incorporates a two-to-many relation, where each `line` referenced two instances of `point1`, and each `point1` could reference any number of `lines`. The `WHERE` statement checks that line endpoints connect to form a closed polygon. In many-to-many relations, an INVERSE clause is required so both sides of the many-to-many relation can be defined. In the example, a many-to-many relation occurs if a line may be part of multiple polygons, as is commonly the case in boundary representation solids.

EXPRESS solves the many-to-many problem by using the INVERSE relation. We redefine the previous `line` example and `polygon` above:

```
ENTITY line;
  point_ref : ARRAY[1:2] OF point1;
INVERSE
  loops : SET [0:2] OF polygon FOR edges;
END_ENTITY;

ENTITY polygon;
  edges : LIST [3:?] OF lines;
WHERE
  2_connected : (* check that the line's endpoints
                        are all two-connected *)
END_ENTITY;
```

The loops INVERSE attribute identifies a constrained SET of references, allowing a line to be composed into zero, one or two polygons. The polygon Entity specifies that it must have at least three lines within it. Thus this relation is many-to-many. In general, many-to-many relations are defined using the inverse clause, with both attributes in the inverse relation being constructors.

5.5.8.2 Supertype Constraints

In the general case, when SUBTYPEs are defined from a SUPERTYPE, relations may exist among the subtypes. A person, for example, may be specialized into male and female. It is not usually possible for an instance of person to be both of these types since the two categories are disjoint. On the other hand, a person may also be specialized into a parent and child, such that an instance may be both a parent and child. Supertype constraints allow the definition of rules on instances restricting their multi-type membership. These constraints apply to instances that are categorized as being of one or more SUBTYPEs of a SUPERTYPE.

EXPRESS provides constraints for the SUBTYPE clause to make these different cases explicit (the EXPRESS documentation calls them both constraints and operators). They are as follows:

ONEOF—defines that the set of SUBTYPEs are mutually exclusive. An instance may be of only one SUBTYPE (as in the male and female subtypes).

AND—used to compose both operands, by logical conjunction; an instance may be of both or all SUBTYPEs.

ANDOR—defines that there is no rule and that an instance may belong to any subset of the SUBTYPEs. If no constraint is specified, ANDOR is assumed as the default. (This is in contrast to other object-oriented languages where ONEOF is the only option.)

These subtype constraints can be useful in product modeling because they allow definition of the different combinations in which Entities may be classified. Examples of ONEOF are spousal relations (husband, wife) and the types of materials used in construction (timber, steel, concrete). Examples of ANDOR might be type of mechanical equipment, where a set may be any mixture of electrical, piping, or air-handling. AND is used only in cases where there are multiple classifications of some high-level Entity type. An example of AND use follows:

```
ENTITY mechanical_part
SUPERTYPE OF (AND (power_part,handling_part));
...
END_ENTITY;

ENTITY power_part
SUPERTYPE OF (ONEOF(fluid_powered,electrical_powered,powerless));
...
END_ENTITY;

ENTITY handling_part;
SUPERTYPE OF (ONEOF(air_handling,liquid_handling,communication));
...
END_ENTITY;
```

In the above type structure, a mechanical_part always includes a power_part and a handling_part. This is a powerful feature, especially useful in product modeling classification. However, there is no direct implementation in existing programming languages, and developers implementing STEP interfaces must develop their own conventions for dealing with it.

5.5.8.3 Abstract Supertypes

A semantic constraint that can be expressed in NIAM and other data modeling languages is that a supertype can only consist of the members of its subtypes. That is, it can have no members of its own. In Figure 5.10, the TOTAL constraint expresses the fact that all pass-thru instances must be of type flexible-pass-thru or rigid-pass-thru. In the example of mechanical parts above, we may wish to require that all mechanical_parts have both a power_part and a handling_part, e.g.,

```
ENTITY mechanical_part
ABSTRACT SUPERTYPE OF (AND (power_part,handling_part));
```

```
FUNCTION distance(p1, p2 : point) : REAL;
(* compute the distance between two points *)
END_FUNCTION;

FUNCTION normal(p1, p2, p3 : point) : vector;
(* compute normal of a plane given three points on the plane *)
END_FUNCTION;

ENTITY circle;
  Center : point;
  Radius : REAL;
  Axis   : vector;
DERIVE
  Area   : REAL := pi * radius ** 2;
END_ENTITY

ENTITY circle_by_points;
SUBTYPE OF (circle);
  p1, p2, p3 : point;
DERIVE
  Radius    : REAL := distance( p1, SELF\circle.center);
  axis      : vector := normal(p1, p2, p3 );
WHERE
  Not_coincident : (p1 <> p2) AND (p2 <> p3) AND (p3 <> p1);
  Is_circle  : (distance(p2,SELF\circle.center) = radius) AND
                 (distance(p3,SLEF\circle.center) = radius);
END_ENTITY;
```

Figure 5.21: Example of code showing EXPRESS syntax as an algorithmic language.

5.5.9 PROGRAMMING CONSTRUCTS

EXPRESS incorporates a large set of language constructs to define the expressions for FUNCTIONs, WHERE clauses and DERIVE clauses. These are similar to those provided in standard programming languages, such as C. FUNCTIONs are used as repeatedly called routines in the definition of complex derivations or rules.

The whole syntax for EXPRESS is not provided here, but an example of a small program is shown in Figure 5.21 to indicate the style of the language. In addition, EXPRESS incorporates many useful functions that facilitate the easy definition of such rules presented in Section 5.5.7. Of particular significance are

In: an infix operator that tests membership in some aggregate. The right-hand operand is checked to determine if the left-hand operand is a

	member. Returns TRUE if it is a member, FALSE if it is not and NULL if either operand is missing
Like:	a string matching operator that returns TRUE if the left operand is a substring of the right operand, FALSE if it is not and NULL if an operand is missing
Typeof:	a function that returns a set of strings of all the types of which the parameter element instance is a member
UsedIn:	a function that takes as an argument an Entity and a string containing an attribute name and returns a set of all the Entity instances that refer to the attribute

Figure 5.21 shows the outline definition of two FUNCTIONs (not fully defined) that are used to derive parameters from a circle that is defined by three points: a center and a begin- and end-point. From these, the WHERE clause derives `radius` and `axis` and checks that the points are not degenerate. EXPRESS also incorporates a query language, allowing data carried within an EXPRESS schema to be interrogated, compared and extracted.

5.5.10 COMBINING MODELS

EXPRESS facilitates the definition of abstract constructs that can be "imported" into other models for use. The two forms of importation are the USE FROM statement, which identifies a schema from which a list of Entities is to be imported into the current schema, and the REFERENCE FROM statement, which is used to reference Entities for use as attributes in the current schema.

```
USE FROM geometric_model
      (faceted_Brep,shell_based_wireframe);
REFERENCE FROM geometric model
      (manifold_surface, AISC_steel_spec)
```

The difference between these two forms of importation is that USE FROM allows subtypes and all other types to use the imported elements. This is done in a manner similar to the facilities in most programming languages, such as the include statement in C and C++. The REFERENCE FROM statement only allows use of the elements as attributes.

5.5.11 EXAMPLE

Figure 5.22 reproduces a standard example from the EXPRESS manual that shows a simple schema for defining a genealogical tree. It first defines three types: an array called `date`, a function for computing a person's age called `years`, and an enumerated attribute called `hair_type`. These are used to define `person` that has three variables for defining their name, a birthdate, a `hair_color` attribute and a relation to a set of the person's children. `Person` is subtyped into `husband` and `wife`. The recursive attribute `children` allows a `person` to refer to other `persons` that are their children. The INVERSE relation of `children` is named `parents`, so that any relation of a person to their children also carries the INVERSE relation of child to parent. This example utilizes the simpler concepts of EXPRESS.

5.5.12 BUILDING MODEL EXAMPLES

The STEP generic resources are available to represent standard geometric Entities. They will be reviewed in more detail in the next chapter, where we will again look at the arc and bounded plane examples.

This section presents the core wall example that was initially defined in Section 3.2. It is presented incrementally, so that various aspects of its definition can be discussed as we proceed. It is useful to compare this EXPRESS model with the roughly parallel NIAM model presented in Section 5.4.2, which has similar intended semantics. Since EXPRESS does not execute WHERE

```
SCHEMA example;

TYPE date = ARRAY [1:3] OF INTEGER;
END_TYPE;

FUNCTION years(d : date) : INTEGER;
(* computes an age to the current date from d *)
END_FUNCTION;

TYPE hair_type = ENUMERATION OF
    (brown,
    black,
    blonde,
    redhead,
    gray,
    white,
    bald);
END_TYPE;

ENTITY person
  SUPERTYPE OF (ONEOF(male, female));
  first_name    : STRING;
  last_name     : STRING;
  nickname      : OPTIONAL STRING;
    hair_color  : hair_type;
  birth_date    : date;
  children      : SET [0 : ?] OF person;
DERIVE
  age : INTEGER := years(birth_date);
INVERSE
  parents : SET [0 : 2] OF person FOR children;
END_ENTITY;

ENTITY female
  SUBTYPE OF (person);
  husband : OPTIONAL male;
    maiden_name : OPTIONAL STRING;
WHERE
 WI : (exists(maiden_name) AND EXISTS(husband)) OR
          NOT EXISTS(maiden_name);
END_ENTITY;

ENTITY male
  SUBTYPE OF (person);
  wife          : OPTIONAL female;
END_ENTITY;

END_SCHEMA;
```

Figure 5.22: The standard example of an EXPRESS model.

clauses and DERIVE computations, the EXPRESS model below conveys these features in words rather than pseudo-code.

```
SCHEMA core_wall_model;
USE FROM geometry (Brep,drawing,polyline,polygon);
USE FROM plumbing (plumb_entity);
USE FROM electrical(elect_entity);
USE FROM hvac(hvac_entity);
USE FROM floor_ceilings(floor_entity, ceiling_entity);
```

```
(* type definitions *)
TYPE resB = REAL ; (* resistance measured as reciprocal of
                  C, measured in BTU/sq.ft.*)
END_TYPE;

TYPE area_in = REAL; -- area in square inches
END_TYPE;

TYPE distin = REAL; -- linear distance unit in inches
END_TYPE;

TYPE UB = REAL; (*coefficient of transmission in
                        BTU/hr/sq.ft.  *)
END_TYPE;

TYPE boundary =
      SELECT(wall,floor_entity,ceiling_entity);
END_TYPE;
```

This model externally references various geometric models that are not included here, using the USE FROM syntax. It also refers to external mechanical systems elements that use the wall as a chase area. To deal with boundaries, it also uses floor and ceiling Entities. Some types are defined for some general measurements that in practice also have standard definitions within Part 41. However, we define them here explicitly and assume they will be resolved with supporting generic resources at a later "interpretation" stage of model development. A SELECT type is also defined, to group the different Entities that might bound a wall.

```
ENTITY wall
      SUPERTYPE OF (ONEOF ( core_wall );
      geom          : wall_geom;
      opening       : OPTIONAL SET [0:?] OF o_object;
      abuts         : LIST [0:?] OF boundary;
      DERIVE hs     : UB  := "derive hs for wall from hs
                             of segments and openings";
WHERE 3D_shape_consistent :="Brep consistent with
                             segments and openings";
INVERSE
   wall_abutted_by : LIST[0:?] OF boundary FOR
                                    wall_abutted_by;
END_ENTITY;

TYPE wall_geom (ABSTRACT SUPERTYPE);
      3D_shape     : Brep;
      flr_plan     : drawing;
      elevation    : drawing;
      WHERE drawing_consistent :="plan and elevation are
                             consistent with Brep";
END_TYPE;
```

A generalized concept of wall is defined, which has a reference to a single ABSTRACT wall_geometry that includes three generic geometric views, for plan, elevation and solid model. The wall references a LIST of o_objects—openings in the wall. The wall has a DERIVED attribute for energy flow that is defined at this level so it can be applied consistently to any wall subtype. Wall also has references to a select type called boundary that defines the different Entities that bound this wall. It carries an INVERSE attribute backreferencing the other boundary Entities that may abut this one. Wall is then specialized into core_wall, as one general type of wall construction. Notice that a designed wall may be defined as wall or as

core_wall. It may be defined first as wall, prior to determining its construction. Later, it may be redefined as a core_wall, when its construction type is decided.

```
ENTITY o_object
      ABSTRACT SUPERTYPE OF (ONEOF (filled_hole,hole));
      region       : polygon;
      r-value      : resB;
INVERSE
      part_of      : wall FOR opening;
DERIVE area : area_in: area_in := "compute area of polygon";
      hs   : UB     := "derive hs from r of filler and area";
END_ENTITY;

ENTITY filled_hole
   SUBTYPE OF o_object
      ABSTRACT SUPERTYPE OF (ONEOF (door,window));
      solid        : Brep;
      pattern      : symbol;
      r-value      : resB;
END_ENTITY;

ENTITY door;
END_ENTITY;

ENTITY window;
END_ENTITY;

ENTITY hole;
END_ENTITY;
```

The o_object is a generalized opening referenced by the supertype wall. It is ABSTRACT, meaning that only instances of its subtypes are allowed and ONEOF, which requires an instance to be of only one of the subtypes. It has an INVERSE attribute allowing any opening to reference the wall it is in. The filled_hole is an ABSTRACT subtype of the o_object; it includes geometric definitions of the filler and references the type of filler. It also carries a thermal resistance. Door and window are defined toward the bottom. Though they add no new attributes of their own, they inherit all the attributes and relations from opening and filled_hole, including the INVERSE attribute and thermal properties.

```
ENTITY core_wall;
      pass_thru    : OPTIONAL LIST [0:?] OF p_object;
      segment      : SET [1:?] OF s_object;
WHERE
      routing      := " pass_thrus only intersect core_walls";
      disjoint     := "all openings and segments are pairwise
                         spatially disjoint";
      coverage     := "all openings and segments cover the wall
                         elevation";
END_TYPE;
```

The core_wall is a subtype of wall. Thus it receives all the references to o_objects and geometry and references the bounding Entities. Core_wall references pass_thrus and segments. It has three constraints defined for it, restricting the possible combinations of data it can carry. They define the legal geometric definitions of a wall and constrain where pass-thrus may go.

```
ENTITY p_object
      pass_thru_entities : LIST OF pass_thru;
```

```
      center       : polyline;
      3D_shape     : Brep;
END_ENTITY;
```

The p_object is a generalized Entity that references other systems passing through the wall and represents their geometry in a way needed by the wall definition. It represents a single pass_thru and carries a polyline and a Brep representation of it.

```
ENTITY s_object;
      core         : core_type;
      insul        : OPTIONAL insulation;
      surf1        : OPTIONAL LIST OF surface;
      surf2        : OPTIONAL LIST OF surface;
      region       : polygon;
 INVERSE part_of   : core_wall FOR segment;
 DERIVE area       : area_in := "compute area of polygon";
      offset1      : distin := "derive thickness to side 1";
      offset2      : distin := "derive thickness to side 2";
      hs           : UB := "derive the thermal flows for this
                    segment from core, insulation and surface Rs";
END_ENTITY;
```

The s_object is a central construct. It defines the construction of a wall segment as a core (with optional insulation) with a sequence of surfaces on both sides. It carries an area and two offsets derived by summing the core offsets and surfaces on each side. It includes the polygonal region it occupies in the core_wall elevation. It computes an area and intermediate resistance for this region of the core_wall. The types of core construction and surfaces are not detailed; they would be subtypes of core_type and s_object, respectively.

```
ENTITY insulation;
      thickness    : distin;
      r_value      : resB;
 INVERSE part_of   : s_object FOR insul;
END_ENTITY;

ENTITY core_type;
      r            : resB;
      offset       : LIST [1:2] OF distin;
      3D_shape     : OPTIONAL Brep;
      INVERSE part_of    : s_object FOR core;
END_ENTITY;
```

The insulation is defined by a thickness and its resistance. It has an INVERSE attribute to the segment it is in. The core_type has a resistance, two offsets and an OPTIONAL Brep solid model defining it. It and other parts of the segment have been defined with INVERSE attributes, so that they access the wall instances to which they belong.

```
ENTITY surface;
      r_value      : resB;
      thickness    : distin;
      INVERSE part_of_1 : s_object FOR surf1;
      INVERSE part_of_2 : s_object FOR surf2;
END_ENTITY;

END_SCHEMA;
```

The `surface` carries a `thickness` and `r_value`, with `INVERSE` attributes back to the `s_object` it is in. It must carry two such attributes because of the two slightly different Roles it may play.

This EXPRESS model is a direct translation of the semantics of the core wall defined in Chapter Three and elaborated in NIAM in Section 5.4.1. It has not been simplified, generalized, or otherwise "tuned" for long-term use. These issues are taken up in Chapter Six.

5.5.13 SUMMARY OF EXPRESS

From the core wall example, we can see that EXPRESS is a rich language generally capable of representing a broad range of semantics associated with buildings or other products. It allows definition of special purpose types, Entities and relations needed for some area of product modeling. It supports sophisticated conceptual definitions of Entity supertype-subtype lattices whose semantics can be defined through SUBTYPE constraints. It also allows definition of complex attributes, including those that are functions of other attributes, making their derivation explicit. It allows definition of complex rules that specify the conditions required for a model to be consistent. It supports specification of two-way relations, allowing bi-directional access through the model structure and guaranteeing that these relations are consistent, using `INVERSE` attributes.

Some limitations also can be identified. Like most other object-modeling languages, EXPRESS does not distinguish relations between an Entity and its attributes and relations with other Entities. There is nothing equivalent to the TOTAL constraint in NIAM, for identifying dependencies regarding potential deletions. This limitation points out the fact that EXPRESS models are static; there are no mechanisms to add or delete parts of a definition. Also, there is nothing equivalent to the UNIQUENESS constraint on sets of attributes which supports queries. There are numerous other differences. In general, however, EXPRESS is able to depict almost all of the semantic conditions defined for the core wall model in Chapter Three.

Given this introduction to EXPRESS, we turn to EXPRESS-G, which is the graphical subset of EXPRESS. We assess both in more detail at the end of this chapter.

5.6 EXPRESS-G

An integral part of the EXPRESS language definition is a graphical notation called EXPRESS-G. While it was defined as a means to depict EXRESS models, it is recognized as a STEP description method in its own right and is becoming frequently used for defining Application Reference Models.

EXPRESS-G allows easy definition of a major subset of the EXPRESS language. It defines a schema in terms of attributes, types, Entities of various type, and also references elements outside the current schema. It provides means to define relations, including Subtype, Derived and Inverse relations. Constrained elements can be identified, but the constraint rule or clause is not specified graphically. There are several slightly different flavors of EXPRESS-G. Figure 5.23 shows the graphical notation corresponding to one of these flavors. All Types but the Basic Types (Real, Integer, Boolean, String, Logical) are represented as a box of dashed lines. Basic Types are solid. Bars on the right or left denote different types. A box with an enclosed circle indicates a reference to a Type of Entity within another schema or diagram. Relations are represented by edges connecting boxes; Supertype/Subtype relations are shown with thick lines, while other relations are shown with thin lines.

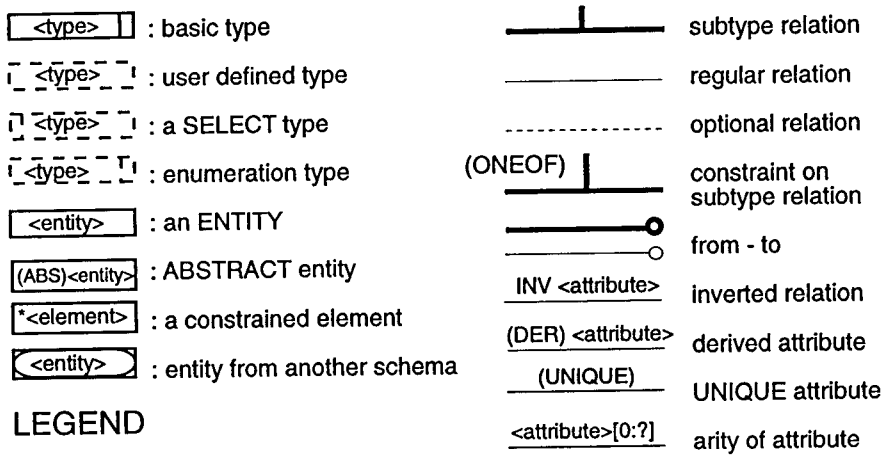

Figure 5.23: The notation of EXPRESS-G. Names of elements are within angle brackets (< >).

In this book, Subtype constraints are shown explicitly. In most other documentation, only the ONEOF constraint is represented, using a "1" to denote it. We denote all three—ONEOF, ANDOR or AND—placing them in parentheses. All relations have a direction denoted by a small circle rather than an arrowhead. OPTIONAL relations are shown with dashed lines (subtype relations cannot be optional). Relations may be qualified with an INVERT constraint or a derivation. INVERTed relations are represented by a single line, with one attribute name shown above the line and the other below it. The arity of relations is shown as subscripts on the relation. Constraints on Attributes or relations are noted with an asterisk (*).

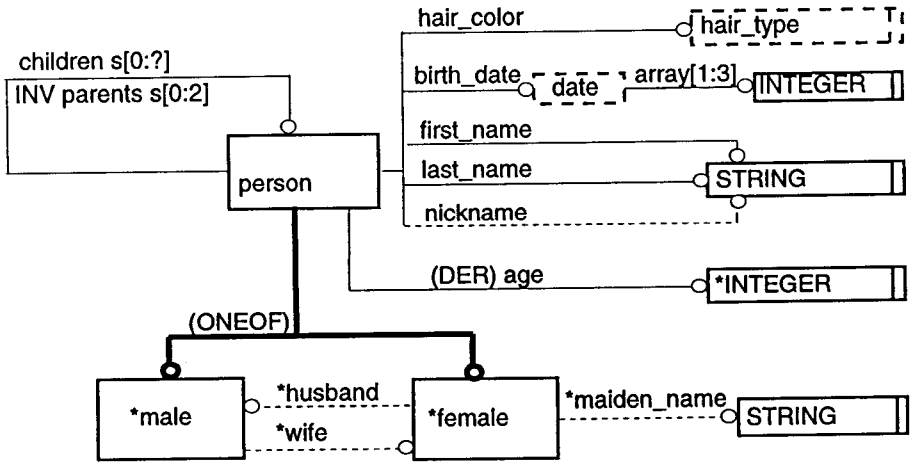

Figure 5.24: The EXPRESS-G representation of the hereditary model presented in EXPRESS in Figure 5.22.

By implementing EXPRESS-G within a graphical drag-and-drop environment, it becomes possible for a user to construct diagrams that, when interpreted by the computer, can be used to automatically generate most of an EXPRESS schema. It becomes a means to graphically define a schema to be used for data exchange. Figure 5.24 presents the example of the EXPRESS schema

defined textually in Figure 5.22, in EXPRESS-G. By checking this figure's correspondence with the earlier example, the semantics of the notation are easily studied. The graphical notation facilitates understanding the overall structure of an EXPRESS model.

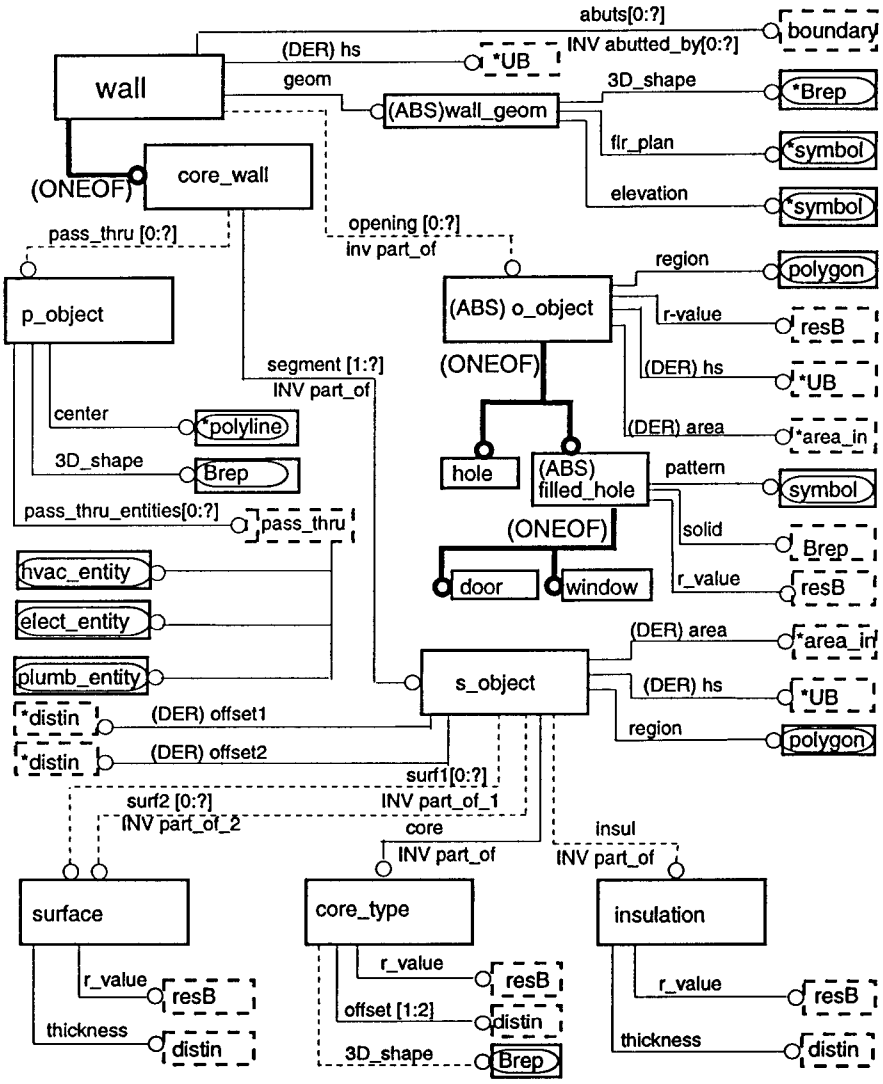

Figure 5.25: The core wall example defined in EXPRESS-G.

Figure 5.25 shows the larger example of the core wall that was developed in the previous section. Presenting an EXPRESS model in this way facilitates its understanding, and allows tracing and debugging. For most people, it is much easier to understand the graphic example rather than its textual version. Again, comparison of the EXPRESS model in Section 5.5.12 with the EXPRESS-G model indicates the semantic expressiveness of EXPRESS-G in relation to EXPRESS.

Once a model has been defined, it must be validated. That is, is the model complete, in the sense that there are no dangling definitions that refer to some Entity or type that is not defined? Are the Entity and attribute definitions that refer to external schemas correct in how the external schemas are referenced? Is the syntax of the model correct? These issues are aided by a variety of software tools.

A small software development community has grown up around the data exchange field. Originally, developers provided IGES translators and IGES file conformance testing software. Software firms, most of which grew out of university or industrial research labs, support the STEP enterprise with a variety of tools. A list of some of the vendors is provided in the Appendix.

The tools include

- EXPRESS compilers that check the syntactic correctness of an EXPRESS model
- cross-references tables that show where each schema, type and Entity is defined and where they are used
- tools that support the automatic generation of EXPRESS-G diagrams with associated browsers, supporting conceptual review
- tools that check if two EXPRESS models are the same or if they differ and how, so that different versions of an evolving model can be carefully compared
- code libraries that can access an EXPRESS schema and Read or Write data in the prescribed format

These tools are described in the Appendix.

5.7 ARM TO AIM INTERPRETATION

The translation of a reference model (ARM) to an interpreted model (AIM) involves rationalizing the ARM specification within the larger scope of STEP Part models. That includes determining the appropriate use of Generic Resources in the model and coordinating the model with the relevant Application Resources. For example, various APs developed in the AEC area should incorporate sufficient overlap to deal with such issues as spatial conflict testing. While initially the emphasis was on specifying ARMs in NIAM or EXPRESS-G, recently some ARMs have also been developed in EXPRESS. In this situation, interpretation means adjusting the already developed EXPRESS model to conform with the Integrated Resources and with the larger STEP environment, defined in other APs.

5.8 PHYSICAL IMPLEMENTATION OF AN EXPRESS REPOSITORY

An EXPRESS schema provides a template for generating a physical implementation of a product model. It defines a structure for storing data describing instances of a building part, assembly or a whole building. While there has been discussion of a range of possible physical format implementations, in practice there have been two types: a computer file that stores the data describing some product or a database that holds the data. Both of these approaches are described below.

The general facilities for developing physical level implementations of an EXPRESS model are defined in STEP Part 22, Standard Data Access Interface (SDAI). Part 22 defines consistent data storing and access mechanisms in terms of their functionality and their programming language interfaces. The interfaces are defined as a library of methods or procedure calls—generally called *language bindings*.

An EXPRESS Part model is a complex structure. One need only review an example model to grasp its complexity. In the core wall model presented in Figure 5.25, for example, what are all the attributes of a window? Looking at window alone, there are none, but in fact, a significant set has been inherited into the window through supertypes. Any interpretation of an EXPRESS model requires scanning up and down the subtype-supertype lattice to put together the structure of an

Entity. In addition, a model utilizes a variety of shared library integrated resources that are in different schemas, requiring going back and forth among multiple schemas.

In order to facilitate easier interpretation of EXPRESS models, a preliminary processing step is to transform the EXPRESS schema into a more easily read dictionary. The dictionary is a compiled form of an EXPRESS model. The dictionary is itself defined in EXPRESS. To give an idea of the dictionary's structure, the Entity and Attribute specifications are given below.

```
ENTITY entity_definition
   SUBTYPE OF (named_type);
   supertypes        : LIST OF UNIQUE entity_definition;
   attributes        : LIST OF UNIQUE attribute;
   uniqueness_rules  : SET OF uniqueness_rule;
   where_rules       : SET OF where_rule;
   complex           : BOOLEAN;
   instantiable      : BOOLEAN;
   independent       : BOOLEAN;
INVERSE
   parent_schema     : schema_definition for entities;
UNIQUE
   UR1: name, parent_schema;
END_ENTITY;

ENTITY attribute
  ABSTRACT SUPERTYPE OF
    (ONE OF (derived_attribute, explicit_attribute,
                                          inverse_attribute));
  name              : STRING;
  domain            :  base_type;
INVERSE
  Parent_entity     : entity_definition FOR attributes;
UNIQUE
  UR1:  name, parent_entity;
END_ENTITY;
```

A structure in this format is defined for each Entity in a schema. For an Entity, the structure is fairly clear. The SUBTYPE carries the name of the Entity. The supertype carries all the supertypes of this Entity class. The attributes carries a list of attributes, as defined below it. Uniqueness_rules defines a set of constraints that apply to the Entity. Where_rules is a set of domain rules for the Entity. Complex indicates whether the Entity is the result of an ANDOR or AND subtype constraint. If TRUE, the Entity is the result of mapping multiple inherited Entities, else it is defined explicitly. Instantiable indicates whether Entity is an ABSTRACT SUPERTYPE. Independent indicates whether the Entity is independently instantiable or whether it relies on a REFERENCE clause. Parent_schema references the schema in which this one is declared. UR1 checks that the name_type is unique within the schema.

The attribute is a supertype of derived_attribute, explicit_attribute, or inverse_attribute. It carries a name and a domain. An inverse relation references the Entities that reference it. Its name also must be unique. Each of the EXPRESS constructs, including the schema, all types, all rules, constructor types of set, list, bag and array are specified in a manner similar to the Entity and attribute above.

Any approach to storing EXPRESS-formatted data must define the equivalency between each construct in EXPRESS and the format in the medium being used. For example, how is an attribute that uses a set constructor to be stored, or how is an INVERSE relation to be stored? These equivalencies can be represented in a mapping table, as described in Section 5.3. An example of a

mapping table is shown in Figure 5.28. EXPRESS vendors have defined and implemented functions that realize such a mapping table for each repository medium to which they have developed interfaces.

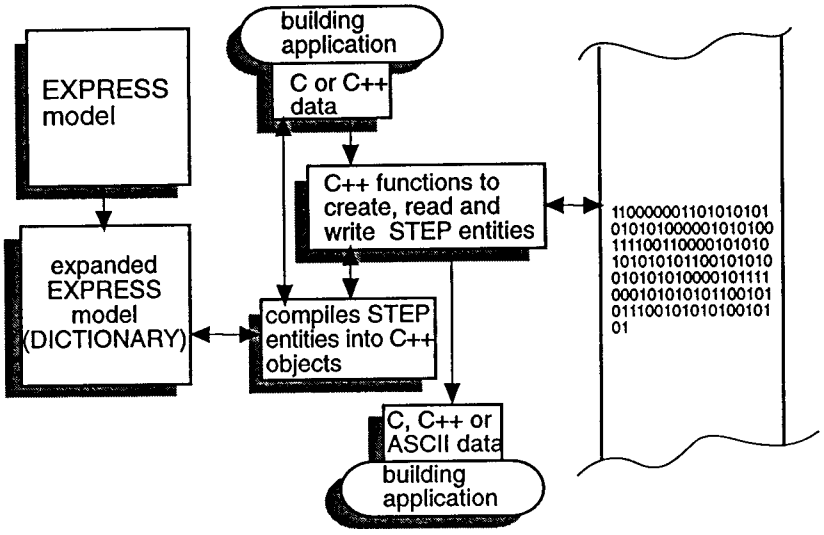

Figure 5.26: The early binding approach to developing a STEP interface.

There are two general strategies for implementing an EXPRESS data repository. These are called *early-binding* and *late-binding* strategies. In an early binding strategy, shown diagrammatically in Figure 5.26, each Entity in the EXPRESS schema for some application protocol is parsed and defined as a corresponding structure in a programming language such as C or C++. For example, an Entity made up of three attributes, a list and a set will be parsed and defined as a C `struct` or C++ object that carries three fields for attributes, a structure for a list and another structure for a set. These structures are created as a preprocessor to compilation, allowing code to be written that both reads or writes the Entities to and from the application and also to and from the storage medium. The read and write code related to the application entity instances is custom to the application. The read and write to the storage medium is provided by language bindings that read or write an Entity instance to or from the storage medium. Since the structures are defined and compiled with integrated reading and writing, they operate quite efficiently. .

In the late-binding strategy, shown diagrammatically in Figure 5.27, the EXPRESS Entities and relations making up a model are identified during the execution of the exchange process. Instead of compiling the elements of the schema, the elements are identified and looked up in the data dictionary for each read and write. For example, using the early-binding example above, the late-binding interface would involve a function call to create an Entity, followed by three function calls to make three attributes, followed by a call to create a list structure, followed by a call to create a set structure. This sequence must be executed for each instance encountered. The advantage of late binding is that the functions used to read or write to/from a format can be defined once and used for any model. The model Entities can be easily revised. With early binding, custom functions must be written for each model Entity.

A particular implementation issue is the supertype constraints on inheritance. These identify a range of Entity types that may be defined as combinations of subtypes—using ONEOF, ANDOR, and AND. At the implementation level, each possible combination of subtypes is explicitly defined. For example, if a supertype "A" has subtypes "ANDOR(B, C, D)", then the following

types are allowed: "A+B", "A+C", "A+D", "A+B+C", "A+B+D", "A+C+D", "A+B+C+D". "A" is inherited into all subtypes and all combinations of "B", "C" and "D" must be generated. There is no way to automatically generate these types. They must be defined by a programmer.

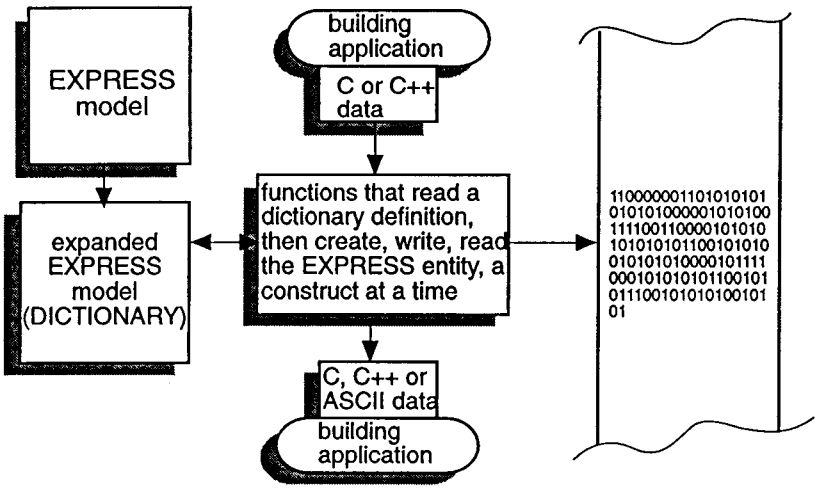

Figure 5.27: The structure of a late binding SDAI implementation.

5.8.1 TEXT FILE IMPLEMENTATION

The most prevalent implementation—and the one for which most tools have been developed—is using a text file. A file-based repository may be developed using either early- or late-binding strategies. Part 21 of the STEP standard fully specifies how an instance in any EXPRESS format should be organized in a physical file. Thus it defines how a dataset of Entity instances in EXPRESS format should be laid out. It uses a free format not based on columns and specifies a file composed of two sections, a header section and a data section.

The header section has three parts: (1) a file description part that provides an informal description of the file contents; (2) file registration information consisting of the filename, timestamp, author, organization, processor that generated the data, release status, and related information; and (3) the file schema format, defined for the EXPRESS AIM used, and any Application Interpreted Constructs used with the AIM. The second section consists of Entity occurrences, formatted according to a syntax uniquely mapping the EXPRESS schema into locations within the dataset.

Comments may be inserted into the physical file data section, delineated by /* and */. The file may include print format statements and "white-space characters" for enhanced readability. These are ignored when the file is parsed for translation.

Each construct and construct composition used in EXPRESS can be defined to specify a format for the EXPRESS schema on a file. Only some of the EXPRESS constructs are needed, and others are mapped onto a common format. The table in Figure 5.28 defines how each construct is mapped. The right side of the table shows the formats used in the physical file.

These equivalencies are built into functions that are then embedded into application programs that READ, WRITE, QUERY and CHECK the data stored on a file, using the conversions shown. The CHECK functions are used to validate the data carried within a model. Typical CHECK functions include validating that the attributes of an instance are of the correct types, validating that the

instance carries values for all attributes that are not OPTIONAL, and validating that there is an appropriate pairing of INVERSE attributes.

EXPRESS CONSTRUCT	PHYSICAL FILE FORMAT
schema	---ignore---
array	list
bag	list
list	list
set	list
select	---ignore---
Entity	Entity
Entity as attribute	Entity name
Entity as supertype (no instance)	---ignore---
Entity as supertype	Entity
integer	integer
real	real
string	string
binary	binary
Boolean	enumeration
logical	enumeration
enumeration	enumeration
derived attribute	---ignore---
inverse	---ignore---
procedure	---ignore---
function	---ignore---
constant	---ignore---
remark	---ignore---
rule	---ignore---
where rule	---ignore---

Figure 5.28: The mapping table for EXPRESS constructs mapped to physical file constructs.

File level implementations are sometimes called clear text encoding. They are appropriate for writing out a model from one application program and reading all or part of it into another application program. Modifications of the data within the same file are not possible, because of the lack of a way to adjust the file's storage allocation.

Recently, a new non-standard text-encoding scheme has been developed using SGML, the Web-based data exchange language.

5.8.2 DATABASE IMPLEMENTATION

There have been a few database implementations that support storing and retrieving data in an EXPRESS-consistent format. Various implementations are possible. Most rely on an encoding and database representation of each EXPRESS language element as defined in the EXPRESS dictionary. These can be concatenated into high-level database calls to Create, Write or Read schema Entity instances.

Databases provide a higher level of functionality to the physical implementation of an EXPRESS model, which is reflected in the SDAI interface facilities. These include

- *Session operations:* start event recording, stop event recording, open session, close session
- *Transaction facilities:* start transaction in read-write access, start transaction in read-only access, promote to read-write access (from read-only access), commit, abort
- *Model access and management:* open repository, close repository, rename model

Database interface libraries have been developed and are marketed for a variety of commercial databases, including the Oracle relational database, and the ObjectStore, Versant, and ROSE object-oriented databases.

5.9 CONFORMANCE TESTING

Conformance testing is the evaluation of an implementation to determine whether it conforms to the standard. It begins with the development of a detailed set of tests and test purposes, while the AIM is still under development. Each implementation of an AP is required to have associated conformance criteria that can be used to validate any implementation of the AP. The test criteria include the options and variations that are to be exercised during conformance testing of an AP implementation.

In general, conformance requires satisfaction of the following criteria:

1. The information requirements of the ARM are preserved in the implementation. This includes all combinations of Entities and their attributes.
2. All AIM Entities, types and associated constraints are supported. Treatment of options shall conform to the AIM.
3. Only the constructs defined in the AIM are recognized and included in their implementation.

The implementation must also specify whether the the AP is required to be complete or whether subsets are allowed. If subsets are allowed, they must be defined.

These criteria are used to create a test suite for any implementation of the AP. This test suite is to allow testing laboratories to define and apply executable tests during conformance evaluation. The emphasis is on making the tests auditable and easily replicated. All controls are provided and standardized by STEP (with the exception of PIXIT, for which guidance in its definition is given), thus assuring regularity to the highest standards possible.

Guidelines for development of test suites include

1. Identify all Entities that can exist in the ARM without being a child of another Entity; these become test Entities.
2. Identify the correspondence between ARM and AIM Entities and verify that the requirements of all ARM Entities are satisfied.
3. If attributes have enumerated values, there should be tests for each possible value.
4. If attributes are of SELECT type, then each of the possible types allowed should be checked in the tests.
5. If an attribute if OPTIONAL, then the test suite should include cases that both include and omit the optional attribute.
6. Test cases should cover all combinations of types allowed by the supertype constraints.

There is a notion that an AP may be divided into *levels*, where a level is a subset of the overall AP. This provides the option to define partial implementations of the complete protocol. Each conformance level can be considered a miniature AP. This allows each AP to identify subsets of the protocol that can be considered "complete" for some purposes.

5.10 THE EVOLUTION OF STEP

ISO-STEP is one of the largest standards efforts ever undertaken, encompassing many areas of industry with worldwide scope. This is only practical if many different STEP activities are undertaken in parallel. A majority of the Parts identified in Figure 5.3 have on-going committees, which are either in the initial process of generating a version 1.0 AP, or are revising an existing one.

Across such endeavors, many aspects of STEP are evolving in parallel. At the base facilities level, the procedures involved in developing and gaining approval of APs are being refined to make them simpler wherever possible. At the same time, improvements in conformance testing are being sought. With over five years of practical experience using EXPRESS and EXPRESS-G, there are plans to extend and revise it to respond to known shortcomings. In this respect, STEP is different from other standards efforts. For example, in the early days of computing, there was no agreement on the coding scheme for characters. Specifically, there was the EBCDIC character coding used by IBM competing with the now standard ASCII coding. Well-developed working solutions existed and the standards issue was which alternative to select. All character-based devices now rely on ASCII coding as a fixed non-changing standard. STEP is not a fixed, non-changing standard. In this regard, STEP is more like a very large, related set of development efforts.

The ISO 10303 activities making up the STEP enterprise are largely funded through support of various interested organizations. Some are private, such as Boeing, General Motors, Bechtel and the large Japanese construction firms. Others are public organizations, such as the National Institute of Standards and Technology in the US and the various European Union organizations. Many small organizations, including civil engineers, software houses, and universities are also members. These organizations support the activities by providing staff time to develop, review and test proposed APs throughout their development process. Additionally, the standards are under public control and are not proprietary. In some ways, it can be viewed as the "United Nations" of industry standards.

We have covered much intellectual territory in the presentation of STEP, and there are other aspects that we will analyze in more detail in future chapters. It is important to gain a broad picture of all these dispersed activities, because effective contributions—either within the STEP framework or outside it—are difficult to undertake when such large but often poorly understood efforts in data exchange are taking place.

5.11 REVIEW OF EXPRESS AND EXPRESS-G

Here, we return to consider the role and functionality of EXPRESS as a data modeling language for product data exchange, especially for the building industry.

Product models are influenced by various interests. One intellectual goal of product modeling is the representation of all the information used in the conceptualization, design, construction or maintenance of a building—in other words, the creation of a semantically complete model. This goal is one of conceptual modeling, capturing the knowledge of domain experts. People holding this view accept that no application can yet use or manipulate all of the information they want to represent, but they contend that eventually applications will be developed to cover these aspects. By developing a rich model, new applications may be realized more quickly. We might call this intention the "idealistic view". A different influence emphasizes that the goal of a product model is to provide data exchange between applications that exist today. As new applications are devel-

oped, the building model can be adapted to deal with them. Besides, as new applications are developed, they are likely to take unexpected forms and structures that may invalidate any effort to anticipate them. This view takes as its goal the easy mapping of data between applications. We might call this pull the "pragmatic view".

If we take the goal of building product modeling to be a practical one—that of facilitating the use of software applications in support of building activities throughout the building's lifetime—then the goal needs to be closer to the pragmatic intention. The basic reason for data exchange methods is to simplify the integration of computer applications into the workflow associated with a building task. Elaborating the definition of a building model increases its complexity; it may require a number of Entities that are derived or otherwise part of the model, but that are not now used. Thus the idealistic view adds issues and complexity to the pragmatic view of integration.

The design of EXPRESS was based mostly on pragmatic intentions. Its members are mostly practical engineers interested in exchanging what can be represented in current procedural programming languages, especially object-oriented languages such as C++ and Java. It has not emphasized conceptual modeling concepts (such as relations) or logic programming (and inference making). Thus it supports well the programming concepts now used to produce commercial application software.

The one area where EXPRESS includes some notion of conceptual modeling is also the most problematic in its application. The supertype constraints lead to both semantic errors in ARMs and also implementation headaches. Meaningless combinations of types are sometimes inadvertently allowed. At other times, implementations are limited and omit types that should be allowed, because they rely completely on the skill of a software programmer to implement the possible types correctly.

EXPRESS has been used by different Part Committees to define and implement a large number of complex Application Models, all listed in Figure 5.3. As these models have been developed, the inevitable restrictions, awkward aspects and limitations of EXPRESS have become visible. One issue is proper use of the rich semantics. For example, under what conditions should a Set constructor be used and when should a List be used? What is good practice for defining Basic Types in the definition of attributes? Guidelines for good practice have emerged as more models have been developed and the strengths and limitations of the language, as demonstrated in use, have become apparent.

Some areas of global functionality are missing. EXPRESS has been conceived as a format for exchanging data between two applications, with an implicit focus of using a file repository. However, there are many cases where a more permanent repository is desired, allowing multiple updates and incremental changes to the model data. Because physical implementations of databases have not been fully addressed, some issues associated with database implementations have not been adequately considered. How to communicate deletions and/or additions are two specific examples. A related difficulty is that EXPRESS was conceived as a format for exchanging data between two closely related applications using a single schema. It was not defined to support translation between unlike applications that need to utilize different schemas. Recent efforts to add mapping facilities, reviewed in Chapter Eleven, are aimed both at alleviating the static structure of EXPRESS and at allowing translation between heterogeneous types of applications.

A related difficulty is that EXPRESS has been conceived as a format for exchanging data between two applications with the same functionality as a file translator. Because physical implementations of databases have not been fully addressed, some issues associated with this kind of implementation have not been adequately covered. Some of the missing functionality include

- the ability of the user to select subsets of a dataset to send to an application

- the ability to check the consistency of the model over time as updates are made
- the ability to deal with concurrency control when multiple people are updating the model at the same time, and
- the ability to implement larger models encompassing multiple applications, such as those needed for buildings

Many of these issues go beyond the development of a building model *per se*, and suggest the need to address a broader set of issues—what might be called a modeling framework. Some of the issues pertaining to a modeling framework are addressed in later chapters.

Another limitation of EXPRESS is that the functions, rules and other relations that are defined within an EXPRESS model are not executable. The rules are currently used for specifying to "the translation writers" the relations that data is supposed to reflect. Some SDAI libraries support execution of Functions, Rules and WHERE and DERIVE clauses, but, in general, rules or clauses must be checked by the test suites developed for translators to and from the model.

Overall, the issues raised here are detailed ones, made apparent by the success and wide use of EXPRESS as a product model specification language. The evolving challenge will be to maintain its simplicity and clarity, while extending its functionality.

Currently, EXPRESS-G is used as a beginning ARM language, prior to developing more detailed models in EXPRESS. Throughout, it is used as an abstraction of EXPRESS, facilitating high-level organizational review and comprehension (which is how it will be used in the later chapters of this book). In these roles it works quite well. However, if EXPRESS-G is to be relied on for developing Application Reference Models by domain experts—not software engineers—it needs to offer better ways to express constraints and relations. These aspects of a model are not just technical issues best handled by implementers, but a basic part of defining the domain knowledge embedded in an application. As a result, less complete ARMs are defined. New data modeling languages have been developed and are in widespread use in other fields. Advances are continuously being made to NIAM and ER. Especially noteworthy in this regard is the Unified Modeling Language (UML), which is also designed as a pragmatic, implementation-oriented modeling language. It, however, has several semantic constructs not directly represented in EXPRESS-G.

5.12 THE STEP SYSTEM ARCHITECTURE

Before looking at examples of STEP models using this technology, it is worthwhile to assess the technology on its own and in terms of its own goals, as well as those we have defined earlier.

The notion of an implementation-independent standard, with alternative physical implementations, is based on the analogy that there may be different language compilers that allow implementation of a process on different machines. In the analogy, the semantics of the language are used to specify a process having multiple implementations. Similarly, EXPRESS allows the definition of a data representation that is then implemented in different media and storage environments.

STEP has made a conscious decision to partition the UoD of all data exchange into much smaller areas, which are oriented around clusters of software applications, using the Application Protocol concept. However, as Application Protocols grow in number and their overlaps and interrelations become apparent, several product areas have called for another kind of integrated resource, one that provides a framework for how different APs should be organized. So far, such framework models have been proposed for shipbuilding, process plants and for buildings. There are many

issues regarding this new addition to the STEP system architecture that remain to be resolved. Some of them will be discussed in Chapter Eight.

5.13 NOTES AND FURTHER READING

The European efforts that were initiated parallel to or after IGES include SET [1989], CAD*I [Schlechtendahl, 1988], VDA-FS [1987] and EDIF [1985]. STEP was viewed as a pan-European effort to subsume these individual efforts. A review of these efforts is provided in Chapter Four of Bloor and Owen [1995].

The ANSI-SPARC database effort heavily influenced the STEP architecture. It is presented in Tzichritzis and Klug [1978]. The STEP system architecture and development methods are presented in two reports by Danner and Yang [1992a, 1992b], Burkett and Yang [1995] and also Bloor and Owen [1995]. The data modeling languages used in STEP include IDEF1x [1985] and NIAM. An early presentation of NIAM is [Verheijen and Van Bekkum, 1982] and another is Halpin and Nijssen [1989]. A much revised recent presentation of NIAM is Halpin [1995]. Translators that map an ARM language into EXPRESS are described in Chandhry [1992] and Poyet [1993]. An example of a tool that translates NIAM into a relational database schema is presented in De Troyer [1989].

The official EXPRESS document is ISO DIS10303 Part 11 [1991]. Another excellent presentation of EXPRESS is Schenck and Wilson [1994]. Its genesis can be traced from PDDI [1984]. STEP development methods are presented in Danner and Yang [1992a] and [1992b]. Details of the STEP implementation procedures have been omitted. The document specifying these procedures is ISO TC184/SC4/WG4 N34 [1992]. The Standard Data Access Interface for STEP is Part 22 [ISO/WD 10303-Part 22, 1993]. In the US, the National Institute of Standards and Technology maintains a Web site for coordinating ISO-STEP activities:

http://www.nist.gov/sc4/www/stepdocs.htm

There have been many independent efforts to articulate the semantics embedded in data, for use in computer languages and databases—as well as product models. Good surveys of the general area are offered by Hull and King [1987] and by Peckham and Maryanski [1988]. The recent development of Unified Modeling Language (UML) provides another important conceptual model. It is easily accessible in Muller [1997]. Work assessing data models for use in product modeling include Hardwick and Spooner [1987], Fulton and Yeh [1988], Shaw et al. [1989], MacKeller and Peckham [1992], Eastman, Bond and Chase [1991], Eastman and Fereshatian [1994].

5.14 STUDY QUESTIONS

1. Select a catalog that presents information about some building product. Examples might be a catalog for windows, doors, window awnings, exterior material panels. Focus on products whose geometry is defined by just a few dimensions. Also, identify two or more computer applications that might use the information in the catalog. For one line of products within the catalog, develop your own NIAM, EXPRESS-G or EXPRESS model of the product line. The emphasis is to represent the information about the product needed by the applications.

2. Consider the information needed to model a process in a construction schedule. Consider the data in the context of current construction scheduling applications. A reference is Fischer, Luiten and Aalami [1995]. Define a NIAM, EXPRESS-G or EXPRESS model of a general construction process. What kinds of information needs should be included in all such processes?

3. Given a general construction process model, as defined in Question 2, consider the specializations needs for particular classes of construction process. Consider, for example, concrete pouring, painting and excavation work. Again, consider this question in the context of current scheduling programs.

4. There have been few data models developed to define a space within a building. Consider such a model from an architectural design and building code perspective. Building codes in most countries have safety and habitability requirements for residential spaces. Two research efforts related to such an effort are Eckholm and Fridquist [1996] and Eastman and Siabiris [1995]. Develop a NIAM, EXPRESS-G or EXPRESS model for a general residential building space that could be use for checking against building code requirements. Ignore the relations of the space to its surrounding boundaries and focus on the attributes of the space and relations between spaces.

5. Given the space model defined in 4 above, extend it to deal with public spaces in different classes of building types, such as commercial buildings, auditoriums, gymnasiums and restaurants.

6. Compare the constructs in the major modeling language now used in business for data modeling, the Unified Modeling Language (UML), with the constructs in EXPRESS-G. Identify their similarities and differences. Present an argument whether or not UML should be added to the set of "approved" ISO-STEP ARM languages.