VLDB Journal, **4**, 157-191 (1995), Peter M.D. Gray, Editor

Data Model for Extensible Support of Explicit Relationships in Design Databases

Joan Peckham, Bonnie MacKellar, and Michael Doherty

Received August 5, 1992; revised version received, February 14, 1994; accepted June 15, 1994.

Abstract. We describe the conceptual model of SORAC, a data modeling system developed at the University of Rhode Island. SORAC supports both semantic objects and relationships, and provides a tool for modeling databases needed for complex design domains. SORAC's set of built-in semantic relationships permits the schema designer to specify enforcement rules that maintain constraints on the object and relationship types. SORAC then automatically generates C++ code to maintain the specified enforcement rules, producing a schema that is compatible with Ontos. This facilitates the task of the schema designer, who no longer has to ensure that all methods on object classes correctly maintain necessary constraints. In addition, explicit specification of enforcement rules permits automated analysis of enforcement propagations. We compare the interpretations of relationships within the semantic and object-oriented models as an introduction to the mixed model that SORAC supports. Next, the set of built-in SORAC relationship types is presented in terms of the enforcement rules permitted on each relationship type. We then use the modeling requirements of an architectural design support system, called ArchObjects, to demonstrate the capabilities of SORAC. The implementation of the current SORAC prototype is also briefly discussed.

Key Words. Database constraints, semantic and object-oriented data modeling, relationship semantics, computer-aided architectural design.

1. Introduction

The management of design activities in large projects is a multi-level, multi-faceted task (Sathi et al., 1985). Computer tools are needed for the planning, scheduling,

Joan Peckham, Ph.D., is Assistant Professor, Department of Computer Science and Statistics, University of Rhode Island, Kingston, RI 02881, *joan@cs.uri.edu;* Bonnie MacKellar, Ph.D., is Assistant Professor, Math and Computer Science Department, Western Connecticut State University, Danbury, CT 06810, *mackella@wcsud.ctstateu.edu;* Michael Doherty, M.S., is Ph.D. candidate, Computer Science Department, University of Colorado, Boulder, CO 80309, *doherty@cs.colorado.edu.*

monitoring, and analysis of these projects. Methodologies developed for other domains often do not scale well to the complexities inherent in large design environments. Here we describe the use of semantic modeling to design object-oriented databases needed to support design systems. To illustrate the techniques, we show how the prototype Semantic Objects, Relationships, and Constraints (SORAC) system at the University of Rhode Island can be used to specify and automatically generate an object-oriented database for the support of the ArchObjects architectural design system (MacKellar and Ozel, 1991).

A data model with a rich set of built-in relationship types is needed to model the complex and non-standard relationships of the ArchObjects system, because the standard semantic models are inadequate for such applications. For example, relationships that are usually supported by semantic data models include IS-A, aggregation, association, and classification (Peckham and Maryanski, 1988). Relationship types needed in design systems include is-part-of, role, and alternate (MacKellar and Peckham, 1992). Also, much of the information that needs to be represented in a design database is procedural in nature. However, many semantic data models do not provide support for data encapsulation. Their emphasis is on the description of the static structure of the database, and the associated dynamic structure is often not addressed.

Another alternative is to construct an object-oriented database. Object-oriented data models (OODMs; Kim, 1990; Zdonik and Maier, 1990) have become very popular for modeling design applications due to their ability to represent procedural knowledge as a part of an object class definition. Through the use of a sound programming language paradigm, object-oriented models can support the design of objects and methods necessary for the support of complex relationships. This is adequate for the development of unique relationships. However, with the exception of IS-A, the canonical object-oriented data models currently do not offer relationships with built-in semantics. Thus, the developer of a design database would be required to construct ad hoc relationships in support of the application, even when the relationships are of established types. As Rumbaugh (1987) pointed out, this buries the relationship in the code and makes it much more difficult to describe and to understand the structure of the system.

Both semantic and object-oriented data models are lacking in their support for the analysis of the semantics of complex relationships. Built-in relationships are useful in that they provide a vehicle for the careful analysis of database semantics. Since semantics can take the form of intra-object and inter-object constraints, they can have wide-ranging effects on the database. Design tools that support analysis and modification of these semantics can prevent errors in the logical design of complex schema.

Another problem arises when attempting to construct a data model for the support of design databases. While the relationships necessary for these systems possess an agreed upon structure and set of core semantics, they are not completely standard. Even the semantics of frequently used design relationships can easily

vary from application to application. Thus, the relationships presented to the data modeler should be generic in nature, but extensible through a menu of semantics to provide the exact relationship desired. We illustrate such relationship types in later sections.

Sathi et al. (1985) chronicled the difficulties involved in the development of constructs for the design of complex applications and provided a framework for several representation layers. These layers represent a mapping from the specific domain design interface providing constructs for the design of a particular application to the more general implementation layer, which contains the low level data structures used for semantic knowledge representation.

For example, in the architectural domain, a part relationship may be used to model the association between objects and their components. This relationship is presented to the database designer on the domain layer, and then mapped to the semantic layer, which contains a more generic model of objects and relationships that is not application dependent. HAS-PART may be represented internally as an association of part components, where association has a specific, but more generic, meaning and structure. Additional generic semantics expressing the structure and behavior of the relationship can also be expressed here. An example is a cardinality constraint of 1-N on the links of the HAS-PART relationship, specifying the maximum number of objects of one type that may be related to an object of another type.

This information is then mapped to the epistemological layer, where a description of the generic semantics of the modeling constructs is provided. The exact definitions of cardinality and inheritance semantics, for example, are given here. The update behavior of the relationships, indicating how the system will maintain constraints under changes, can also be given on this level. On the logical level, this information is then translated into a set of assertions about the general structure and behavior of the relationships, which is then mapped to the implementation of the system. These layers parallel the levels of the SORAC system, from the upper level application-dependent design layer to the low level object-oriented implementation of the database (Section 6).

Sathi et al. (1985) also addressed the insufficiency of standard relationships for particular application domains and the need for conflict resolution in a complex design to assure system consistency and correctness (Peckham et al., 1995). They provided early documentation of the need for active semantics describing the propagation of activity in a complex set of interconnected relationships, and the ability of relationships to provide availability of information at a distance, from one location to another in the system, issues that we consider important and have addressed in our research.

Rumbaugh (1987) argued strongly for the explicit incorporation of relationships within the object-oriented data modeling paradigm and provided a paradigm for the support of relationships using a C++ type of language. Batory and Kim (1985) identified several relationships, and the associated semantics that are necessary for the design of VLSI CAD (computer aided design) objects. This was one of many

articles documenting the necessity for the extension of semantic modeling constructs to manage design databases. Using a particular design database application, Foo and Takefuji (1990) described the relationships and update semantics needed for the maintenance of their VLSI design database.

Eastman et al. (1991) developed EDM, a data model for engineering product databases. This employs several design oriented, built-in relationships with semantics expressed as constraints in the form of logical expressions. Using these constructs, high-level semantics can be expressed over several relationships to address both local and global schema design, and design integration.

Nguyen and Rieu (1991) addressed the issue of semantic complexity in design objects in their data model SHOOD. Their data model includes only IS-A and part relationships, and they separate semantics from structure to implicitly model other kinds of relationships. Our approach is to permit a greater variety of structural relationships which the schema modeler can define and tailor to a specific domain. Although their work has many of the same goals as ours, they do not present a detailed view of update actions implied by the object semantics.

In another OODB, called ADAM (Diaz and Gray, 1991), relationships are defined as objects, permitting encapsulation of relationship semantics that are viewed in terms of structural dynamic aspects. This refers to the ways in which state changes may occur, including method definitions and derived data. Once defined as objects, relationships may be incorporated into an IS-A hierarchy, permitting users to create specialized relationships. Since user-defined semantics are supported in this approach, it is very compatible with the ideas presented in this article.

Kim (1989) investigated the semantics of composite objects extensively. In his system, ORION, bi-directional relationships are only provided at the implementation level, in terms of backwards pointers. References are distinguished initially on the basis of non-composite vs. composite references. A composite reference is similar to a part relationship. Non-composite references can have no additional semantics associated with them, whereas composite references may be exclusive or shared, and dependent or independent. Kim's view of composite object semantics has much in common with our view of part relationship semantics. ORION does not, however, provide built-in semantics for other types of relationships, such as collections, derivation relationships, or specialized relationships such as adjacency. In our model, relationships play a central role and are as important as objects.

Our work is related to that in the active database community. Morgenstern (1984) provided equations for the expression of database constraints, and algorithms for the mapping of the constraints to database actions maintaining the constraints. Gehani and Jagadish (1991) provided facilities for the specification of constraints and triggers upon objects in their object-oriented database, Ode. The constraint construct includes actions to be performed upon violation of the constraint, thus including the update semantics we advocate. Relationships are not used as a construct for expression of update semantics; the semantics are encoded within the involved objects.

Albano et al. (1991) addressed the modeling of objects, relationships, and constraints through the use of of an object-oriented language employing an object and relationship data model. They have identified the update actions to be monitored due to the possibility that those actions may violate particular constraints. Support for the maintenance of constraints is provided through an exception mechanism.

A major problem in object-oriented databases is to correctly specify operations on the objects that will not violate integrity constraints. This is the motivation behind our view that a schema designer should explicitly specify actions to support relationship semantics. Constraint analysis is an approach in which constraints are explicitly specified using a formal language (Urban, 1989; Urban and Delcambre, 1990). CONTEXT (Urban et al., 1992) is a tool for the explanation of constraints to assist the schema designer in specifying propagation actions. The schema designer must specify the operation; CONTEXT then identifies constraints that would be affected by the operation, along with possible propagation actions. In SORAC, the schema designer directly specifies the database actions through the selection of built-in enforcement rules, and then interacts with a schema checker to determine schema correctness and consistency. An advantage to Urban's constraint analysis method is that the database semantics are expressed in a declarative manner and can be stored as part of the database for future reference. In SORAC, database semantics are expressed through update rules that can be maintained in a similar way, although we do not currently do this.

Other approaches to the problem include the following: Ceri and Widom (1990) and Widom and Finkelstein (1990) provided specification and analysis of update semantics over relational constructs using extensions to SQL. Although explicit relationships are not used, this can be viewed as the implicit definition of relationships between relational tables through the expression of constraint and update expressions. In the similar approach of Bouzeghoub and Metais (1991), a semantic model was used to capture the database constraints, and map them automatically to implemented update rules in an object-oriented database. However, in their system, update rules are not directly specified, but are inferred from the constraints.

This article presents the SORAC data modeling tool for the definition of complex objects and relationships. In Section 2, we compare the capabilities of semantic and object-oriented data models in the context of databases for support of design systems. In Section 3, we employ the ArchObjects architectural design system (MacKellar and Ozel, 1991; MacKellar, 1992) for examples of the types of relationships that are frequently needed for design databases. In Section 4, we describe several SORAC relationship types, which we believe are necessary for architectural design systems, ArchObjects in particular and design databases in general. In Section 5, we show how an illustrative subset of the ArchObjects relationships can be constructed using the relationships developed in Section 4. In Section 6, the implementation of SORAC is briefly described. The result illustrates the advantages of the integration and augmentation of object-oriented and semantic modeling techniques to meet the

data modeling challenges provided by current advanced applications.

2. Relationships

2.1 Interpretations of a Relationship

There are several ways that a relationship between the objects of a database schema can be viewed. For example, we can think of a relationship as a pathway for the flow of information or data. This is clearly illustrated by the Cactus system (Hudson and King, 1988). In this system, object and relationship specifications include names, types, and a direction for the values that flow between related objects.

In our view, a relationship is also a construct over which constraints specifying the relative states of objects can be expressed. For example, existence constraints and cardinality constraints express the relative existence and the relative number of objects, respectively, that may be related to an object of another type. SORAC *enforcement rules* enforce constraints and are thus associated with relationships:

Definition 1. An *enforcement rule* specifies the exact operational means for the maintenance of a database constraint.

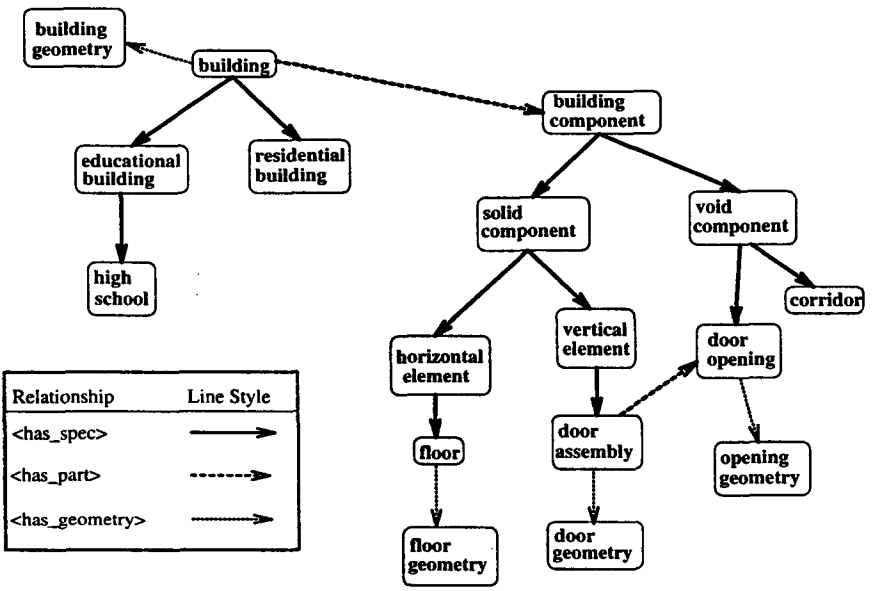
A relationship can also reflect the sharing and organization of the data, thus facilitating the proper association and sharing of operations among the objects of the database schema. This is clearly illustrated by the usual expression of IS-A within most object-oriented data models. In the ArchObjects system, the database schema organizes design objects using several types of hierarchies (Figure 1). As described in Section 3, ArchObjects also organizes design objects according to another relationship, called `<has_role>`, that represents functional role(s).

2.2 Semantic Versus Object-Oriented Relationships

Several authors have compared and contrasted semantic and object-oriented data models (Hull and King, 1987; Kim, 1990). The focus is usually on the structural modeling capabilities of the semantic models, as opposed to the integral structural and operational modeling capacity of the object-oriented models. Each is characterized by a different relationship style that affects the modeler's ease of relationship specification.

Semantic data modeling systems have tended to present relationships between database objects using a more explicit construct than object-oriented systems. Figure 2 is an example of a semantic presentation of a relationship, in which OT_1 and OT_2 represent object types, and the line connecting these two types expresses a relationship between them. Thus, in this graphical mode of expression, relationships are modeled as edges and object types are modeled as nodes. In the Entity-Relationship model (Chen, 1976), relationships are also differentiated from objects, but they are similar since relationships can hold data.

Figure 1. Building component hierarchy

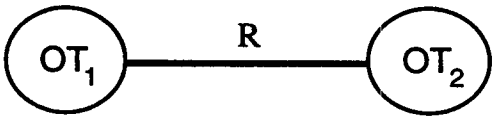


These semantic presentations are convenient since the relationship provides a good receptacle for inter-object constraints and inter-object activity. In addition, bi-directional relationships are easily represented. This paradigm is also convenient since it provides a basis for the clear presentation of hierarchical and deeply nested structures. For example, consider the architectural design hierarchy presented in Figure 1. There are several object types and relationship types present. Since the semantics of each relationship is very different, this can be a confusing structure to analyze. Viewing the relationship as a separate, typed construct provides a powerful mechanism for the specification, analysis, and understanding of the update behavior of the database.

Semantic data models also present collections of built-in relationships having clearly specified semantics. These semantics can include relationship structure, constraints, and enforcement rules. When the modeler defines a relationship to be of a given (built-in) relationship type, (s)he is automatically specifying the semantics as well. Some recent extensions (Dogac et al., 1986; Barsalou et al., 1991; Doherty et al., 1993) include the automatic generation of code that will maintain enforcement rules associated with relationship constraints.

In the object-oriented modeling paradigm (Kim, 1990; Zdonik and Maier, 1990), object types and their associated methods are the basic constructs for defining the structure and behavior of database objects. The use of IS-A to support inclusion polymorphism, in which a type's methods can be applied to all of its subtype's instances, provides a powerful structural and procedural paradigm for modeling

Figure 2. Semantic relationship



complex applications. However, with the exception of IS-A, the object-oriented models do not have built-in relationship types. As a result, many authors (Kim et al., 1989; Zdonik, 1990) have argued for the augmentation of the object-oriented models with other built-in types.

The canonical object-oriented data models do permit definition of complex nested objects; related object types can be defined as attributes within a new object type representing the relationship. For example, in Figure 3, we have given a C++ type of definition of a Door class containing attributes `latch` and `opener`. `Latch` and `opener` are two attributes which are of types `Latch*` and `Opening_Mechanism*`, respectively. The * defines a pointer to objects of type `Latch` and `Opening_Mechanism`.

The `Door` type, through its implementation, serves as the container of the semantics of the relationship between the three types, `Door`, `Latch`, and `Opening_Mechanism`. This technique can be used to model hierarchical situations. For example, a part relationship hierarchy can be defined on each level in the same way that `Door` was defined above. C++ code implementing each type in the hierarchy would be written, including the semantics managing the part links just below it.

There are, however, three disadvantages associated with this relationship representation. First, it is difficult to clearly represent propagated inter-object activity within the database. In this representation, we have code representing separate levels of the hierarchy, but no clear representation of the hierarchy as a whole. Second, many times it is more natural and consistent with the real world to think of relationships as different from objects. Objects usually correspond to real world objects; relationships usually represent abstract connections between objects. Third, the code for inter-object behavior is written for each individual type that is created. There is no reuse of the semantics relating types to each other.

This analysis of the semantic and object-oriented presentations of relationships indicates that each modeling technique has strengths and weaknesses. The differentiated object and relationship presentation of the semantic model is convenient for representing and analyzing deeply nested structures and the inter-object activity upon these global structures. The object-oriented presentation of relationships is convenient for the specification of complex object types as well as the definition of methods. However, the object-oriented model is lacking in that it does not provide strong support for global analysis of the inter-object activity within the database. Each is capable of expressing relationship structures, but for some purposes the expression is awkward.

Figure 3. Door class definition

```
Class Door: public opening
{
public:

/*Constructors*/
Door() ;
Door(Latch *initial_latch, Opening_Mechanism *initial_opener) ;

/*Destructor*/

~Door() ;

/*Attributes*/
private: /*The private attributes latch and opener*/
    Latch *latch;
    Opening_Mechanism *opener;

Public:/*The public access methods for Latch and Opening_Mechanism*/
    Latch *Get_latch() ;
    Opening_Mechanism *Get_Opening_Mechanism() ;
    void Has_Latch (Latch *new_latch) ;
    void Has_Opening_Mechanism(Opening_Mechanism *new_opener) ;
};
```

In the next sections, we discuss the modeling of the ArchObjects architectural design system and illustrate the role that semantic modeling techniques can play in object-oriented data modeling systems. In the current implementation, ArchObjects is based on a set of standard relationships, but in reality the semantics of these relationships are embedded into individual methods defined upon the various types. It is very difficult to ensure correctness and consistency of updates with this type of ad-hoc implementation. Therefore, the idea of built-in semantic relationships, as defined in earlier semantic models, is quite desirable. This would free the schema designer and method writers from worrying about maintaining the semantics of such relationships.

The reasons for adding built-in relationship semantics are as follows:

1. To support reuse of relationship semantics. The same relationships are used over and over. We don't want to have to keep building the same semantics into individual methods, but would rather have them enforced automatically.

2. To facilitate evaluation and enforcement of complex constraints, both at the trigger level and at the rule evaluation level.

3. To allow analysis of relationship interactions during schema design. Explicit representation of relationships facilitates automated schema design tools that check for correctness of a schema design and tools that enforce consistency of the complex constraint set.

SORAC is a modeling tool that imposes a semantic object/relationship view on an object-oriented DBMS (database management system). By requiring the schema designer to specify the enforcement rules for each relationship type, the designer is forced to clearly specify relationship behaviors. SORAC supports the designer in this task by automatically generating the underlying class definitions and enforcement rule methods. To demonstrate the capabilities of semantic object/relationship modeling, a data model for ArchObjects was developed.

3. ArchObjects

This section presents an overview of the ArchObjects system and explains the data modeling capabilities necessary for the definition of an integrated object-oriented database to support the system. ArchObjects is an intelligent design assistant that functions in the domain of architecture. Design verification services are provided by evaluating the design against a set of design codes, particularly legally required rules such as fire code. Relationships between objects function as a central organizing concept within the knowledge representation which is essentially object-oriented, both structurally and in terms of behavior. The original Prolog implementation did not have explicit support for relationships or behavior across relationships; this discrepancy between our conceptual model of ArchObjects and its actual implementation caused numerous problems and inefficiencies. It was decided that further development of this system would be facilitated if it took place within an object-oriented data model that provides built-in support for relationship semantics. Correctly maintaining semantic choices would then be the responsibility of the data model rather than individual method programmers. SORAC presents an ideal environment for accomplishing this goal, by supporting built-in, customized relationships.

The current version, ArchObjects2 (Roberts, 1993), is an intermediate step. It implements part of the SORAC model in C++. In particular, it includes relationships as first class objects, support for queries on relationships, and enforcement rules. Although this version was not generated by SORAC, it is very similar to the type of schema that would be produced. The motivation for this version, in fact, was to verify that the type of schema produced by SORAC would be suitable for a design system such as ArchObjects. We have, in addition, generated a preliminary schema for ArchObjects using SORAC (Vora, 1992). This schema is not as complete as ArchObjects2 and does not include query support, but it verifies that SORAC is capable of generating this type of schema.

3.1 ArchObjects Conceptual Model

ArchObjects stores a semantic representation of each architectural object. One problem inherent to the design domain is that representations of objects that are otherwise perceived as similar may vary quite a bit. This means that representation schemes that depend heavily on standardized sets of attributes for objects of the same type will not be suitable for representing this type of data. Object instances are much more complex and detailed than the object types of which they are members. Thus, there is a need to define the semantics of relationships between instances of design objects and to support these semantics within the data model.

The set of relationships in ArchObjects consists of:

- `<has_instance>` : An object instance is connected to its type through this link. The instance inherits attributes and methods from the type.
- `<has_spec>` : This is a relationship between two object types, and means the same thing as IS-A.
- `<has_attr>` : This relationship asserts that a particular value is associated with a particular object. In ArchObjects, attributes are atomic.
- `<has_geometry>` : This is a binary relationship between a design object and its geometry. It is discussed in more detail below.
- `<has_member>` : This is a relationship between a set or collection object, and the individual objects that comprise the set.
- `<has_part>` : This models the fundamental hierarchical relationship of design objects. For example, a single floor may be composed of a set of rooms. Each room has a set of walls as components, and at least one entrance. An entrance may include a door assembly, which is composed of a frame and a door.
- `<has_role>` : This is a hierarchical relationship between a design object and a description of its role in the design. The object, in addition to having a primary type, also has a particular role inherited from the role type. Such functional components cannot be easily incorporated into the primary object type hierarchy since the functionality may cut across several object types. For example, not all doorways function as protected exits (i.e., fire exits) and, in addition, other objects such as windows may in certain circumstances function as exits. Therefore, a separate type hierarchy based on functionality is part of the knowledge representation. As with `<has_instance>`, methods and attributes are inherited via this relationship.
- `<adjacent_to>` : This is a relationship between an object and the parts that are adjacent to it in the design. Two objects are said to be adjacent if they do not participate in a `<has_part>` relation, and the distance between them is less than a specified threshold.

```
rel(R, adjacent, Obj1:Thing, Obj2:Thing) IF
   (distance(Obj1, Obj2) < d_MAX AND
   NOT(has_part(Obj1,Obj2) OR has_part(Obj2,Obj1))
```

- `<connected_to>` : This relationship is similar but requires that two objects have overlapping volumes

  `rel(R, connected, Obj1:Thing, Obj2:Thing) IF`

  `(Volume(Obj1` $\cap$ `Obj2)` $> V_{MIN}$.

The semantics of these relationships are considered to be standard across the system although, in the first version, these semantics were implemented in an ad hoc fashion within object type methods.

Geometric classes and symbolic classes exist in separate class hierarchies. Each object instance has a symbolic aspect and a geometric aspect. The part hierarchy in the geometric model is strongly isomorphic to the part hierarchy of the semantic model in the sense that

1. each object `Obj1` participates in at most one `<has_geometry>` relationship with a geometry object, and
2. `has_part(Obj1, Obj2)` if and only if `has_part(Obj1Geom, Obj2Geom)`.

From a logical design perspective, the geometric model forms a complete subsystem which is capable of functioning in isolation from the semantic model. Separating the two models at this point simplifies the design and results in a consistent system organization (Dube and MacKellar, 1992). Since relationships are central concepts, database update actions are expressed at the lowest level as insertions or deletions of relationships between objects. For example, a method call to create a new instance of a type would generate a series of low level database update actions to insert a room instance, connect a wallset instance to the room instance, and set the maximum height attribute to a default value of 12 feet.

We have found that it is quite difficult for the schema designer to keep track of all possible relationship interactions without an automated modeling tool such as SORAC. Instead of hand-coding the relationship semantics and hoping for correctness, the data model should present a menu of supported semantics to the schema designer. Schema design tools can then analyze the designer's choices for consistency and the system can then generate code to maintain the enforcement rules.

3.2 ArchObjects2 Implementation

To address the concerns presented in the previous section, we undertook a project to update ArchObjects and port the symbolic model to C++ and the SORAC paradigm (Roberts, 1993). Since this project was begun before the SORAC implementation was completed, we do not automatically generate the schema. Instead, we produced by hand the type of implementation that SORAC would have generated so that we could test the behavior of the enforcement rules. In fact, one of the goals of this implementation was to investigate update rules with an eye towards including them as an integral part of the planned solid model (Dube and MacKellar, 1992).

Figure 4. ArchObjects2 relationship classes

```
class HasPart : public Root {              // Basic HasPart abstract class
     System * const SystemPtr;             // Address of System object
     const int HasPartId;                  // HasPart's unique id
     Object * const CompositePtr;          // composite object
     Object * const ComponentPtr;          // component object
     ...details omitted...
protected:
     HasPart ( Object &, Object & );       // Initialize parameters
     ~HasPart ();                          // Disconnect from PartTable
public:
     static HasPart * create ( Object &,   // Conditional creation
        Object & );
     int destroy ();                       // Destruction control
     ...details omitted... };
class HasPart13 : public HasPart0 {        // Wall/WallSection class
     int Side;
     int FromLeft;
     HasPart13 ( Object & a, Object & b, int i )
        : HasPart0 ( a,b ), Side ( i ) {}
     int validSection (const Wall &, const Wall Section & );
public:
     static HasPart13 * create ( Object & a, Object & b );
     ...details omitted...};
class ProtectedExit : public Role0 {       // ProtectedExit class
public:
     ProtectedExit ( const System & s, const int i ) :
        Role0 ( s, "ProtectedExit", i, 0 ) {}
     ...details omitted...};
```

The objective was to translate the relationships described in the definition of ArchObjects to a SORAC database that includes:

- Architectural object classes,
- Application-specific has-part and has-role relationship classes,
- Enforcement rules to maintain existing constraints, and
- Basic facilities for processing queries over current design contents that include provision for querying over specific architectural objects.

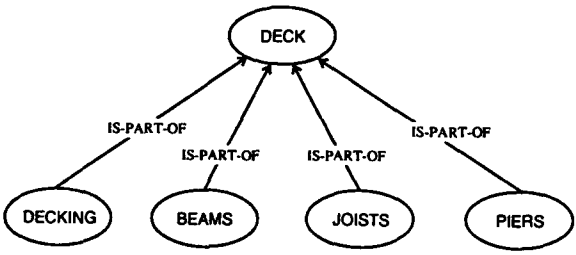
3.2.1 Relationships. Since a major feature of SORAC is its emphasis on built-in relationships, the manner in which relationships were implemented was a critical aspect of the implementation. C++ provides direct support for the <has_attr>, <has_inst>, and <has_spec> relationships that are required by ArchObjects. There are no native C++ constructs for representing either the <has_part> or <has_role> relationships. Thus, the corresponding base relationship classes from SORAC were duplicated; relationship classes derived from these provide constraints and behavior details that are appropriate for particular relationship instances. Figure 4 shows part of the base HasPart class, and two derived relationship classes, one for part relationships between a wall and a wall section, and the other for a role relationship between objects and the role "Protected Exit."

3.2.2 ArchObjects2 Application Module. While a separately compiled symbolic framework contains the relevant built-in features of SORAC, individual applications are constructed by creating an application module through the use of derived classes. The architectural classes selected for inclusion in the application module are familiar objects that could be used to provide a simple design for a one story building. Classes include DoorAssembly, Building, Floor, Ceiling, Door, DoorFrame, WindowFrame, and TrapDoor, as well as the derived relationship classes. The principal information embodied within an object class is the definition of data members, and the enforcement rules to enforce creation (insertion) and deletion constraints that are specific to objects of that type.

Most constraint enforcement occurs when <has_part> relationships are created. One example is the Wall/WallSection part relationship shown in Figure 4. When this relationship is established, the user must supply information indicating the side of the wall to which the wall section is attached, and the distance from the left end of the wall to the beginning of the wall section. This information will be stored in data members associated with the relationship class. As part of the creation process, the WallSection's size and location are used to ensure that the proposed WallSection does not overlap with those previously related to the wall.

Another example of an enforcement rule is provided by the Room/WallSection part relationship. When an internal WallSection is created, it must be associated with a Room. This is an example of a creation constraint. A related deletion rule is required to maintain the constraint, which specifies that, if the <has part> relationship is deleted, then the WallSection must also be removed. Another rule specifies that if a Room object is deleted, all part relationships in which it participates are deleted as well. This results in a cascading set of deletions from the Room object, to the part relationship object, and finally to the WallSection object. When update rules are buried inside methods, as in the first ArchObjects, this type of propagation behavior is very difficult to foresee. Explicit representation of update rules makes it much easier for the schema designer to catch this behavior.

Figure 5. Definition of DECK object class



4. SORAC: A Semantic Object/Relationship Model

SORAC is a prototype data modeling system that accepts semantic object and relationship definitions, and maps them automatically to C++ implementing an ONTOS (1991) database that maintains the specified constraints and update semantics. SORAC's conceptual model is semantic in that it includes built-in relationship types to be chosen by the data modeler. For example, since a part relationship is frequently employed for design applications, a predetermined relationship type having the structure and semantics of this relationship is offered by SORAC. The structure of an IS-PART-OF relationship is shown in Figure 5. The relationship connects a DECK type with the associated DECKING, BEAMS, JOISTS, and PIERS types. IS-PART-OF identifies the built-in relationship used. As outlined in this section, a menu of optional enforcement rules is provided to the designer who then specifies the exact semantics of the relationship.

SORAC provides two schema design interfaces: Architectural Relationships and Constraints (ARAC; Vora, 1992) and Database Schema Design Tool (DSDT; Dong, 1992), as well as a database generation component, Object Interface Language (OIL; Doherty et al., 1993). ARAC and DSDT permit the designer to model the schema using object types, relationships, constraints, and enforcement rules as the primary modeling constructs. The output from each interface is OIL code, which is then used to map the specification to an ONTOS object-oriented database, including code implementing the enforcement rules. More detail of the SORAC implementation is given in Section 6.

4.1 Relationship Semantics

We define the semantics of a relationship as structure, cardinality, existence, notification, and selection rules:

- *Structure:* Specifies the relationship connections between the types involved in the relationship. For example, IS-PART-OF is a relationship between an owning type and one or more part types.

- *Cardinality:* Specifies the maximal possible instances (objects) of a given type that can be connected to another type over a relationship link. For example, IS-A has cardinality 1-1, thus exactly one supertype object is connected to a subtype object over an IS-A link.
- *Existence:* Specifies the relative presence of objects over relationship links. For example, the existence semantics of an IS-PART-OF relationship between BEAMS and DECK types might state that every DECK object must be connected to a BEAMS object through IS-PART-OF.
- *Notification:* Specifies that the system must notify the end user of all actions taken by the system to maintain an enforcement rule. For example, if the end user deletes a DECK object, and the system responds by deleting all of its parts, then the end user is notified.
- *Selection:* Specifies related objects that are presented to the end user upon querying an object. For example, if a DECK object is queried, then the related part objects connected via the IS-PART-OF relationship would be displayed as well.
- *Inheritance:* Specifies how objects may reuse definitions from related types. Examples are given in Sections 4.1.2 and 5.3.

For each relationship type, some of these semantics are built-in and others are given as options. These relationship types, along with their various options, are to be used as constructors of relationships for individual systems. The schema modeler interacts with the system by selecting a relationship type with built-in semantics. Then optional semantics are chosen from a menu to model the exact characteristics of the domain. Later, we will show how these relationship types can be used to define relationships needed for the ArchObjects system. In this article, we will not dwell on the expression of cardinality constraints and the supporting enforcement rules. (For good coverage of this material in the context of the Entity-Relationship model, see Dogac et al., 1986).

The following sections outline the relationship semantics defined and implemented by the SORAC group. There are two general relationship enforcement rules always supported for all relationships. The general relationship insertion rule states that, for a relationship to be correctly inserted, both participating instances must have already been inserted. The general relationship deletion rule states that if either participating object instance is deleted, the relationship instance must also be deleted.

4.1.1 HAS-ATTRIBUTE Semantics. The fundamental HAS-ATTRIBUTE relationship is supported by the relational and semantic models in the form of aggregation. When the attribute is of a subrange or system built-in type, then the semantics are straightforward. We need only to specify the behavior of the system upon the change of an attribute to an out of range value.

Also needed are additional semantics to specify correct behavior when an attribute is used to refer to other objects in the database. Also, if inverse relationships

are maintained, an SDM-like construct (Hammer and MacLeod, 1981) should be provided to permit the definition of a bi-directional relationship of the HAS-ATTRIBUTE type. That is, the user can specify an inverse relationship through an option. For example, suppose we have a DOOR type that has an Opener attribute of type OPENING-MECHANISM, where OPENING-MECHANISM is an object type. We can define the following to specify the inverse of the relationship Opens between a door and its opener.

DOOR $Opens^{-1}$:HAS-ATTRIBUTE Opener:OPENING-MECHANISM

To be consistent with our philosophy of specifying update behavior at modeling time, we must offer additional supporting semantics indicating the behavior of the objects related through the HAS-ATTRIBUTE relationship. The possibilities were discussed and outlined by Peckham et al. (1989), and thus are not repeated here. Briefly, cardinality, existence, notification, and selection semantics can be offered as in Section 4.1.3 for the IS-PART-OF relationship.

The built-in attribute relationship was not implemented in the SORAC prototype. In both ARAC and DSDT, attributes are implemented directly as C++ class variables. However, a built-in HAS-ATTRIBUTE relationship is interpretable by SORAC's database generation component, OIL. In ArchObjects2, attributes were mapped directly to C++ attributes.

4.1.2 IS-A Relationship Semantics. The IS-A relationship is central to both semantic and object-oriented data models. However, as is the case with many other "standard" relationship types, the semantics of IS-A are not fixed. One of the problems occurs with inheritance. For example, in some semantic interpretations of IS-A, if a revolving door IS-A door, and door has attribute color, then this attribute is also inherited by the revolving door. Thus, if the door object is defined to be blue, then we can also view the revolving door object as having the color blue. Multiple inheritance results when we permit a type to have more than one supertype. If two supertypes of an object both have the same attribute, then techniques for resolution of the names of inherited attributes must be developed.

Due to the various approaches that can be taken with respect to the semantics of IS-A, we approach this relationship just as we do the others that follow. We define a generic IS-A having the core semantics that are essential in any IS-A relationship, and include a menu of semantics which can be additionally imposed upon the relationship.

IS-A has the usual graph structure supporting single and multiple inheritance. The following semantics can be additionally imposed.

1. *Inheritance Semantics:*

   (a) Multiple inheritance (MI): If the modeler chooses this option, multiple inheritance will be supported with built-in attribute resolution facilities.

We will not elaborate upon the means of resolution, but assume a valid technique (e.g., that of Stefik and Bobrow, 1986).

(b) Inheritance suppression (IS): This descriptor is used within a subtype definition to specify the attributes that are not inherited from the supertype.

2. *Cardinality/Existence Semantics:*

(a) Strict mandatory subtype membership (SMSM): This means that every object that is an instance of the supertype must also be an instance of exactly one of the subtypes. In the absence of this selection, this condition is not required. Upon insertion of the supertype object, the subtype object must also be clearly indicated (Hammer and MacLeod, 1981). Deletion of the subtype object requires deletion of the supertype object.

(b) Mandatory subtype membership (MSM): This means that every object that is an instance of the supertype type, must also be an instance of one or more of the subtypes. This is similar to the SMSM choice, except that we are permitting membership in at least one and possibly more that one subtype. Update semantics as in SMSM above are also included.

See Section 5.1 for the definition of the ArchObjects `<has_spec>` relationship using SORAC's IS-A semantics. The SORAC DSDT design interface has implemented semantics similar to MSM and SMSM, providing four choices for connections and rules between subtype and supertype. SORAC ARAC has implemented a more refined set of semantics for inheritance permitting the designer to suppress inheritance of attributes and/or inheritance of relationships. This means that we can specify on the type level if a subtype object will inherit the relationship links specified for its supertype object(s). For example, if REVOLVING-DOOR IS-A DOOR and DOOR is connected to PROTECTED-EXIT via a role relationship, we can specify that REVOLVING-DOOR will not inherit this relationship (cannot serve as a protected exit).

4.1.3 IS-PART-OF Relationship Semantics. The IS-PART-OF relationship is necessary for the support of design databases. However, the semantics for this relationship are not universally established (Kim, 1990; Geller, 1991; Halper et al., 1992). Again, we define a generic built-in relationship having semantics that describe the structure and cardinality of the IS-PART-OF relationship. Other semantics are given as options.

The structure of the IS-PART-OF relationship is shown in Figure 5. DECK is an *owning* object type that represents an object constructed from objects of *part* types DECKING, BEAMS, JOISTS, and PIERS. The cardinality of the relationship

between DECK and each part type is N_i, where $N_i \in [1, .., \infty]$ and is user specified, meaning that each part object may be part-of at most N_i owning objects. The IS-PART-OF relationship is used whenever the exact number and types of parts of a design object are known. The collection relationship described next can be used if the exact components of the design object are not known in advance.

During most of the initial design phases, many of an object's parts may not be described. The rules that determine which part relationships (with possibly empty objects) *must* be instantiated are given by the existence and notification semantics.

1. *Existence Semantics:*
   (a) The following rules can be chosen in support of a constraint stating that a part object must belong to at least one owning object.

   DELETION:

   i. Mandatory Deletion (DM): Upon deletion of the owning object, all part objects associated with it are deleted. Upon deletion of an IS-PART-OF relationship instance between an owning and a part object, the part object is deleted.

   ii. Conditional (DC): Upon deletion of the owning object (or a relationship instance between an owning object and a part object), all part objects which are not participating as a part of any other IS-PART-OF or COLLECTION relationship are deleted.

   iii. Blocking (DB): The deletion of an owning object (or a relationship instance between an owning object and a part object), is denied if it will leave a dangling part object.

   INSERTION:

   i. Mandatory (IM): Upon insertion of a part object, the associated relationship link to an owning object must also be established.

   (b) The following rules can be chosen in support of a constraint specifying that an owning object may not exist without all of its parts.

   DELETION:

   i. Mandatory Deletion (ODM): Upon deletion of one of the parts of an owning object (or one of the relationship instances between an owning object and a part object), the owning object is also deleted.

   ii. Blocking (ODB): Upon the user's attempt to delete a part of an owning object (or an IS-PART-OF relationship instance), the user is notified that this action is not permitted.

   iii. Conditional Deletion (ODC): Upon the user's attempt to delete a part object (or relationship instance between an owing and part object), the user is asked if the associated owning object should also be deleted. If so, the part and owning objects are deleted, otherwise, the deletion of the part object (relationship) is not permitted.

INSERTION:

i. Blocking (IB): Upon insertion of an owning object, information indicating relationship links with parts existing in the database must also be supplied. Otherwise the insertion is denied.

ii. Conditional (IC): Upon insertion of an owning object, relationship links with all parts will be established with data present in the database if identified. When this information is not supplied for all parts, part object stubs will be created for later specification.

2. *Notification Semantics* (DPN): Upon insertion/deletion of an owning or part object (or deletion of an IS-PART-OF relationship instance), the user should be notified of any propagated activity resulting from the action.
3. *Selection Semantics*: Mandatory Selection (MS): Whenever an owning object is queried, then the part objects are also returned. In the absence of this choice of semantics, the owning object will be selected, with only references to the part objects.

The database modeler, on choosing an IS-PART-OF relationship, automatically gets the generic semantics. In addition, the modeler can choose selection semantics, insertion and deletion semantics for each existence constraint desired, and notification semantics for any of the insertion/deletion semantics that specify propagated actions.

The existence and notification semantics were implemented by the ARAC interface, and were also included in ArchObjects2. Selection semantics are not currently supported by the SORAC system due to the lack of an implemented query interface. If implemented, selection semantics could be associated with a whole IS-PART-OF instance, or with individual part links. Automatic generation of a query interface that fully supports semantic relationships is an interesting problem that we have not yet fully investigated.

4.1.4 COLLECTION Relationship Semantics. The COLLECTION relationship is used to specify objects consisting of collections or sets of objects of a given type. If a collection consists of objects of many types, then it is assumed that a supertype encompassing all of the classes included is defined through generalization before the collection type is defined. Another alternative is to have the modeling system perform this task automatically on behalf of the modeler when the collection object is defined. The modeler will, of course, have to enumerate the types included and provide a name for the new generalized type.

The semantics of the collection relationship will differ depending upon the needs of the particular application. For example, there are some modeling circumstances in which we might not wish to delete the collection object if it becomes empty through deletion of its members (Kim, 1990). An example of this is a builder's definition of a deck in which a PIERS object represents a collection of actual resources used for piers. We might temporarily take the pieces of lumber away from the deck for a more immediate project, but we will eventually build the deck,

thus, lumber will be assigned to the deck at a later date. In this case, we wish the empty PIERS object to remain, even though the details of its resources are not yet available.

The cardinality of the collection relationship is always *M:N*, where $M,N \in 1..\infty$. This means that, for each collection object, there may be up to *M* member objects, and each member object may be in up to *N* collections.

1. *Existence Semantics:*
   (a) Member Deletion: The following rules can be chosen in support of a constraint specifying that a collection object may not exist without at least one member in the collection.
      i. Mandatory Collection Deletion (MCD): This means that upon removal (through object deletion or relationship instance removal) of all of the members of the collection object, the collection object is deleted.
      ii. Conditional Collection Deletion (CCD): This means that the user may decide whether to permit the empty collection remain or to delete the collection. The system notifies the user of the situation and permits a run time choice of the appropriate action. (This makes the above stated constraint soft, as empty collections may actually occur.)
      iii. Blocking (BCD): The user is not permitted to remove (through object deletion or relationship instance removal) the last remaining member of a collection. This should not be chosen with the BMD rule.
   (b) Collection Deletion: The following rules can be used to support a constraint stating that member objects cannot exist without the associated collection object.
      i. Mandatory Member Deletion (MMD): Upon deletion of the collection object (or removal of a relationship instance), all member objects associated with it are deleted.
      ii. Conditional Member Deletion (CMD): Upon deletion of the collection object (or removal of a relationship instance), all member objects that are not part of any other existing IS-PART-OF or COLLECTION relationship are also deleted.
      iii. Blocking (BMD): (This should not be chosen in conjunction with the BCD rule). The end-user is denied the ability to delete a collection object that has remaining members.
2. *Notification Semantics:*
   (a) Notification (DCN) This means that upon the deletion of the last object in the collection, the user is notified that it is the last object. The system proceeds as the deletion semantics specify, but the user is notified of any additional actions.

Notice that, if none of the above options are chosen, the system will permit collection objects to be empty, even though there are no remaining members in the collection. This is because there are no actions implied when the last member is removed. The semantics of the COLLECTION relationship type were fully implemented by the ARAC schema design interface.

4.1.5 Derivation Semantics. Derivation semantics are used whenever an attribute of an object is derived from attributes of other (domain) objects. It is convenient to specify these interdependencies among objects as relationships, since the manipulation of domain objects affects other objects that have values derived from them. The relationship semantics (enforcement rules) specify exactly how derived attributes are correctly computed when updates to domain objects occur. In ARAC, derivations are specifiable over all relationship types.

1. *Update Semantics:*
   (a) Virtual (V): The attribute value is never stored. It is computed upon access by the user. Appropriate messages are sent to the user in the absence of data needed to compute the derived value.

If this option is not selected, the attribute is stored and updated in the database whenever modifications to the domain values occur, whenever domain objects are inserted, or whenever a relationship link to a domain object is instantiated. In this case, one of each of the following sets of semantics must also be chosen to determine how the database will behave whenever referenced values are inserted, deleted, or modified.

1. *Existence Semantics:*
   (a) Deletion:
      i. Null (NL): Upon deletion of one of the domain objects, the derived attribute is set to *null.*
      ii. Deletion Denial (DL): The user is denied permission to delete a domain object. The derived object or the derivation relationship link between the domain object and the derived object must first be removed.
2. *Notification Semantics:*
   (a) Notification (MDN): Upon modification of a domain attribute or deletion of an object containing a domain attribute the user is notified of a possible effect upon the derived attribute.

5. Example of Constructed Relationships for ArchObjects

We now show how the relationship modeling needs of ArchObjects can be satisfied by the built-in SORAC relationship types of Section 4. Of particular interest is

the combination of generic and add-on semantics needed to model this application. Some of these semantics were implemented in ArchObjects, to test the functionality of this approach.

5.1 <has_spec> Relationship

The <has_spec> relationship represents specialization. For example, if A <has_spec> B, then B is a specialization of A. The <has_spec> relationship is one-to-many, transitive, and is defined only between types. Inheritance occurs along this relationship, and each object of type A must be connected to an object of type B through this link.

To model this relationship, we use the inverse of SORAC's IS-A relationship type. For example, we can establish a <has_spec> relationship between COMPONENT and FLOOR, CEILING, and WALL types:

COMPONENT <has_spec>: $IS - A^{-1}$ (SMSM) FLOOR, CEILING, WALL

As explained above, the option SMSM indicates that each COMPONENT object must be an instance of exactly one subtype. That is, a component must be exactly one of a floor, a ceiling, or a wall. This mode of relationship specification permits us to specify different levels of a <has_spec> hierarchy with different characteristics, such as SMSM versus MSM. However, in most systems, the specialization hierarchy has uniform characteristics throughout the system. Thus, we can also define the <has_spec> relationship in the following way:

Relationship <has_spec> : $IS - A^{-1}$ (SMSM)
COMPONENT <has_spec> FLOOR, CEILING, WALL

This permits the definition of the <has_spec> relationship between other sets of object types without repeating the specification of <has_spec> in terms of IS-A. This feature is available in the DSDT design interface.

5.2 <has_part> Relationship

Here is a collection of ArchObjects relationship classes specifying the parts of a room class:

```
Room <has_part> Wallset
Room <has_part> Floor
Room <has_part> Ceiling
Room <has_part> Openingset
Room <has_part> Elementset.
```

If we assume that `Wall`, `Floor`, `Room`, `Ceiling`, and `Element` have been previously defined as object types, the SORAC specification of this structure will be as follows:

ELEMENTSET COLLECTION (CMD) ELEMENT
WALLSET COLLECTION (CMD, CCD) WALL
OPENINGSET COLLECTION (CMD, CCD) OPENING

ROOM <has_part> : IS-PART-OF^{-1} (DC,IC) WALLSET
ROOM <has_part> : IS-PART-OF^{-1} (DC, IC) FLOOR
ROOM <has_part> : IS-PART-OF^{-1} (DC, IC) CEILING
ROOM <has_part> : IS-PART-OF^{-1} (DC, IC) OPENINGSET
ROOM <has_part> : IS-PART-OF^{-1} (DC) ELEMENTSET

Here we have used SORAC's IS-PART-OF to construct the ArchObjects <has_part> relationship to model a room and its components. Elementset is an arbitrary and possibly empty set of architectural components of a room. The inverse notation for the IS-PART-OF relationship is important since the insertion and deletion semantics on the owning and part types are not necessarily symmetric. As specified above, ROOM is the owning type, and WALLSET, FLOOR, CEILING, OPENINGSET, and ELEMENTSET are the part types. Specifying that the type of an <has_part> link is IS-PART-OF^{-1} (DC, IC) explicitly states the exact enforcement rules of the relationship. The update rules implemented in ArchObjects2 support this approach (Section 3.2.3).

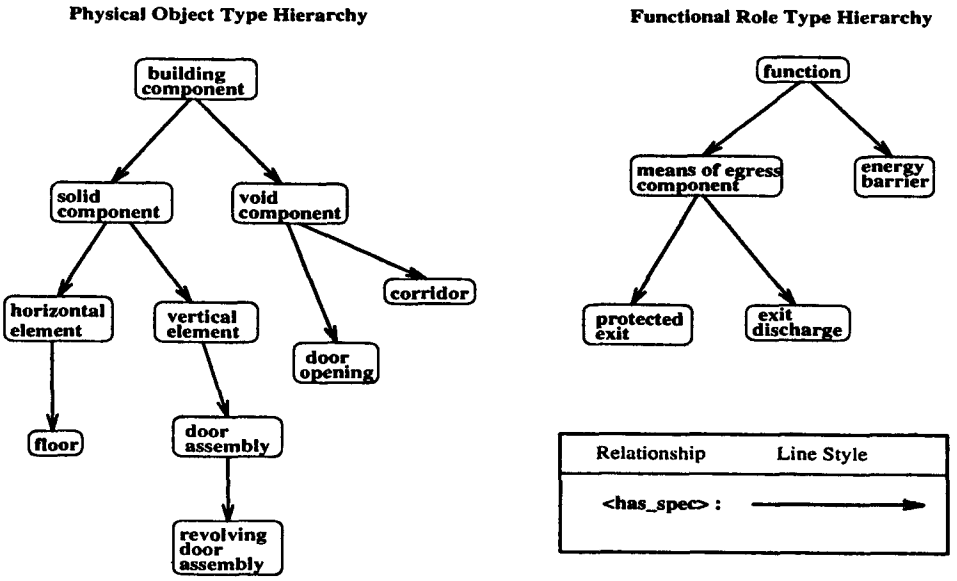
5.3 <has_role> Relationship

As discussed in Section 3, there are two distinct IS-A hierarchies in the ArchObjects system representing design objects and their functional roles. For example, design information such as dimensions and location would make up an instance of type CORRIDOR. However, the corridor object may serve as a means of egress and, thus, have data associated with it in its role as a means of egress. The object type CORRIDOR participates in an IS-A hierarchy of design object types. Similarly, there are many role types (e.g., MEANS-OF-EGRESS COMPONENT) that participate in an IS-A hierarchy of which the most general type is FUNCTION. Thus, we have two somewhat orthogonal hierarchies describing objects and roles (Figure 6).

To model the associations between the objects and their functional roles, we need a means by which we might draw horizontal arcs from one hierarchy to the other. For example, if we know that objects of type CORRIDOR might serve as a means of egress, we could define an association or reference relationship between CORRIDOR type and MEANS-OF-EGRESS COMPONENT, thereby indicating the potential for relationship links on the instance level.

However, this approach is quickly foiled by the complexity of the design environment. Consider an example such as REVOLVING-DOOR ASSEMBLY. A revolving door is a specialization of a door, but may not legally serve as a protected exit. For simplicity, we would like to draw a horizontal line between DOOR ASSEMBLY and PROTECTED-EXIT but, since all properties of DOOR ASSEMBLY are inherited by REVOLVING-DOOR ASSEMBLY, we cannot. Also, we notice that many objects participate in several roles. These subsets of all role types are

Figure 6. Physical object and role type hierarchies



not particularly well organized through the use of the IS-A hierarchy, thus forcing the definition of several horizontal associations between a given object type and several role types.

One solution to this problem is the combined use of the collection and the attribute relationship types. For a given object type, we consider all possible roles in which an object of that type might participate. We then define a HAS-ATTRIBUTE relationship using the most specialized role supertype that characterizes these roles as follows:

DOOR<has_role>:HAS-ATTRIBUTE Role:MEANS-OF-EGRESS-COMPONENT

Thus, we have associated the type DOOR with the role types in which the DOOR might potentially participate. We have named the horizontal link <has_role>, and defined it to be of type HAS-ATTRIBUTE. Additional semantics (Section 4.1.1) might also be specified with the HAS-ATTRIBUTE relationship type. For the <has_role> relationship, we must choose semantics that do not force an individual door object to participate in any role. The relationship is there to support the potential for a door to participate in the relationship. Notice also that if we wish to define a different collection of functional roles for the type REVOLVING-DOOR ASSEMBLY, we can first suppress the inheritance of the Role attribute from DOOR, and then tailor a role set for the revolving door that eliminates the protected exit role.

In contrast to the general SORAC semantics, the ARAC interface does not require use of an attribute relationship to model this connection between the part and role hierarchies. A built-in HAS-ROLE relationship is available, and additional inheritance semantics permit the suppression of relationship semantics from supertype to subtype in an IS-A hierarchy. This was the approach taken in ArchObjects2 as well.

5.4 <has_geometry> Relationship

In the ArchObjects system, there is a special relationship used between the types representing the physical components of a design (e.g., FLOOR, DOOR ASSEMBLY), and objects representing the geometry of the objects. To model this situation, all objects representing physical components are defined to be subtypes of supertype PHYSICAL-COMPONENT. The supertype has an attribute, Solid-Model-Set, that is a collection of SOLID-MODEL.

SOLID-MODEL-SET Collection:COLLECTION (MMD,MCD) SOLID-MODEL
PHYSICAL-COMPONENT <has_geometry>:
HAS-ATTRIBUTE Solid-Model-Set: SOLID-MODEL-SET

The following outlines the semantics of the has_geometry> relationship as defined.

- A PHYSICAL-COMPONENT object can have zero or one geometries, but the collection object should not be present in the case of zero geometries, thus the MCD option is chosen.
- A SOLID-MODEL-SET cannot exist without the corresponding PHYSICAL-COMPONENT object. Thus, semantics enforcing the deletion of the SOLID-MODEL-SET upon deletion of the PHYSICAL-COMPONENT should be chosen for the HAS-ATTRIBUTE relationship. The MMD option takes care of removing the SOLID-MODEL objects. Semantics prohibiting the insertion of SOLID-MODEL-SETS (unless they are attached to a PHYSICAL-COMPONENT object) should also be chosen with the has-attribute relationship.

In addition to the above, a PHYSICAL-COMPONENT's parts can be derived from the parts of the corresponding SOLID-MODEL-SET. This can be specified using the derivation relationship.

6. SORAC Implementation

In this section, we describe the implementation of the current SORAC prototype, with emphasis on the underlying support supplied by OIL (Doherty et al., 1993). OIL provides programmable support for semantic relationships in an object-oriented data model. This is accomplished by permitting the definition of high level object and relationship semantics, and then by mapping these semantics to object-oriented code in a well established and predictable manner.

The SORAC system has two application schema design interfaces (ASDIs), Architectural Relationships and Constraints (ARAC) and Database Schema Design Tool (DSDT). Each ASDI permits the high level design of objects and relationships, and then automatically generates OIL output. Through translation of semantics into OIL, SORAC provides automatic generation of databases having complex relationship semantics, including those needed for systems like ArchObjects.

The following criteria motivated the design of the system.

- The conceptual constructs in the designer's schema appear as addressable entities in the user's view of the database. In particular, relationships must be available as both modeling constructs and database objects.
- The semantics of standard system defined relationships and arbitrary user-defined relationships are equally supported. As illustrated by the ArchObjects examples given above, design systems for different domains require different sets of standard relationships. In addition, it is generally necessary to allow the designer to define new relationships to handle unanticipated special cases.
- The specification and implementation of the enforcement rules that maintain inter-object constraints must not violate object-oriented encapsulation. The implementation of data objects should not be influenced by the relationships in which they may potentially participate.
- The mapping from the conceptual schema to the implementation should be automatic.
- A tool should be available to check the schema semantics for correctness, completeness, and consistency before they are mapped to the implementation.

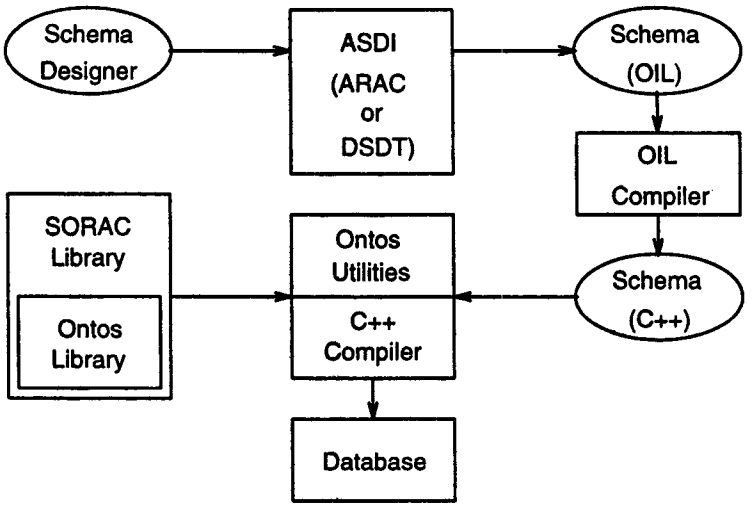
To provide this functionality, OIL maps relationship specifications, expressed in the OIL language, to code that implements a *monitor* construct. This allows objects to monitor the activities of related objects and thereby maintain the semantics of relationships.

6.1 Overview of SORAC System

The organization of the SORAC system is shown in Figure 7. The ASDI represents ARAC and DSDT, and can be viewed as a replaceable module that supports the specific semantics of a particular database domain. The ASDI outputs a schema definition as a set of OIL object type declarations. The OIL compiler generates C++ code compatible with the ONTOS object-oriented database management system (Andrews and Sinkel, 1991; ONTOS, 1991), and this code is processed by ONTOS utilities and the native C++ compiler to generate the required database. The support of relationship behavior is implemented as an extension to the ONTOS Client Library.

The prototype OIL compiler and the SORAC extensions to the ONTOS ODBMS have been implemented on a SUN Sparc station. ARAC and DSDT both have been

Figure 7. SORAC system



integrated with the SORAC system. A query language for SORAC databases has not yet been defined but the database implementation has been tested through a textual interface that allows the user to construct and send messages to any object in the database. Some query language issues have been explored in the ArchObjects2 implementation.

A schema checker that uses semantic information from the ASDIs to interact with the designer to assure correct and consistent schema has been developed (Peckham 1994; Qian, 1994). A graph theoretic representation of the objects, relationships, and enforcement rules is used with properties such as the transitivity of update actions to discover awkward, incorrect, and/or incomplete schema structures such as cycles and conflicting updates. These structures are then presented to the database designer to aid in refinement of the schema specifications.

6.2 OIL Interface

The OIL interface supports the SORAC data model in which a relationship is viewed as an interactive object. The semantics of a relationship are defined by the constraints that the relationship places on the behavior of the participating objects. The interactive nature of relationships in the SORAC data model allows these constraints to be modeled as behavior of the relationship, rather than behavior of the participating objects. The relationships are mapped to objects in the implemented database yielding the benefits of relationships as modeling constructs (Rumbaugh, 1987).

Figure 8. ARAC DoorHasLatch relationship

```
ARAC Specification:

Relationship: DoorHasLatch
    Relationship Type: Has_Part
    Composite Object: Door
    Part Object: Latch
    Enforcement Rules for Deletion of Composite Object:
        DM (Mandatory Deletion)

OIL Specification:

object DoorHasLatch
    {participants
        {object Door composite;
         object Latch part;}
    monitor (composite.delete) // Mandatory Deletion constraint
        {updateif (part != NULL) delete part;}}
```

The OIL language is based on C++ with the addition of *participants* and *monitors*. A list of participants and a set of monitors is what distinguishes a relationship from a data object. The participants list defines the object types that may participate in the relationship and the roles that those object types play in the relationship. Monitors implement the constraints that define the semantics of the relationship. The monitor construct enhances the object-oriented model by allowing relationships to act in response to the operations applied to participant objects. These operations act as triggers that cause the monitor to execute the *enforcement rules* maintaining the constraints. The syntax and implementation of monitors are derived from the *propagator* (Laffra and van den Bos, 1991) which demonstrates that inter-object constraints can be implemented in a manner that does not violate encapsulation. By defining monitors as properties of relationship objects, the SORAC data model combines the benefits of relationships as modeling constructs with the encapsulation of the propagator.

The enforcement rules which define the operation of a monitor are taken from Peckham (1994), where they are defined for the analysis of schema correctness. Since the behavior defined by a monitor's enforcement rules is directly mapped to the behavior of a relationship object, any correctness guarantees made through analysis of the enforcement rules can be assumed for the resulting database. Since the mapping is automatic, the possibility of incorrect translation from the data definition to the database schema is avoided.

Figure 9. DSDT DoorHasOpener Relationship

```
DSDT Specification:

Relationship: DoorHasOpener

Participants:
   Source: Door
   Destination: OpeningMechanism
Constraints:
   Existence: OpeningMechanism depends on Door
   Cardinality: 1_to_1

OIL Specification:

object DoorHasOpener
   {participants
       {object Door source;
        object OpeningMechanism destination;}
   monitor (source.delete) // existence constraint
       {delete destination;}
   monitor (destination.delete) // existence constraint
       {reject;}
   monitor (self.insert) // cardinality constraint
       {if (source.ParticipatesIn("DoorHasOpener")) reject;
        if (destination.ParticipatesIn("DoorHasOpener")) reject;}}
```

6.3 Implementation of Enforcement Rules

The following examples illustrate the specification of relationships in the ARAC and DSDT ASDIs, and the OIL code that the interfaces generate. The ARAC definition of a part relationship named `DoorHasLatch` between a `Door` and a `Latch` is first shown in Figure 8. Since ARAC was developed to support architectural design systems, a built-in and generic has-part relationship type is provided. The maintenance of an existence constraint between the Door and the Latch is assured through the optional choice of the DM enforcement rule. The OIL code that is generated by ARAC directs the generation of lower level code implementing a monitor which triggers on a Delete message to the Door and maintains the constraint by deleting the related part, as well as the relationship.

The definition of the Door and Latch object types can be specified independently of the HasPart relationship, since the semantics of the relationship are encapsulated in the definition of the relationship. Implementation details, such as inverse pointers and triggers, are automatically generated by the OIL compiler and the SORAC library extensions. The OIL interface thus provides an intermediate layer of abstraction, which allows for the expression of complex objects and relationships without the details required by a typical object-oriented implementation.

Figure 9 shows the DSDT definition of a `DoorHasOpener` relationship. The OIL code that is generated shows the meaning of the DSDT existence and cardinality constraints. The existence constraint, specifying that OpeningMechanism depends upon Door, is maintained by deleting the related OpeningMechanism object whenever a Door object is deleted and denying the independent deletion of OpeningMechanism objects associated with Door objects. The one-to-one cardinality constraint is maintained by monitoring the instantiation of relationship instances and denying the instantiation whenever there is more than one Door object associated with a given OpeningMechanism object, and vice versa.

7. Conclusion

In this article, we developed a model that augments the object-oriented model with a group of core semantic relationships, from which domain specific relationship types can be derived. This was done in the context of ArchObjects, an intelligent design database operating in the domain of architecture. Design databases are a good example of the need for incorporating semantic relationships. On the one hand, the object-oriented model is a very natural way to represent a design, because inheritance and specialization allow knowledge to be stated at the highest possible level. On the other hand, much of the understanding of a design in a domain-like architecture is based on the relationships between objects. It is clear that these relationships are more than mere references between objects. In fact, much of the meaning of the design model is conveyed through an understanding of the relationship semantics. Therefore, the relationship semantics should be made explicit, rather than buried in method code. Furthermore, certain relationships seem to be fundamental in a given design domain, and are used over and over throughout the model. The `<has_part>` relationship is a good example of this. It is much simpler if the semantics of these relationships are handled by built-in enforcement rules, rather than re-implemented again and again. It is even more desirable if this set of semantics is not fixed, but can be tailored to the domain by the schema designer.

Future additions to this work include the augmentation of interrelationship semantics. There are possibly many ways in which semantics may be specified over subsets of relationships. These needs should be documented, and a manageable set of these semantics should be formulated to resolve and clearly specify interactions among relationships. We also plan to investigate tools for maintaining and browsing through the update rules, so that the database semantics are accessible once the database has been built. In addition, we are also investigating the role of relationships and enforcement rules in real time database applications (Prichard, 1994) and in forming complex views in design domains.

Further work in the development of query primitives capable of operating over the defined relationships of a given database is also of interest. For example, if a `<has_role>` relationship has been defined to be of type HAS-ATTRIBUTE then, at design time, we know the structure, the semantics, and the name of this relationship.

The design interface should be capable of generating code for the support of query primitives, which can easily operate over the role relationship. In this way, we have generated a schema with built-in update support, as well as meaningful querying tools. Techniques for the generation of this type of support should be carefully investigated and prototypes should be developed to provide a basis for the analysis of the possible approaches.

The area of design is heavily based on relationships that often have a very specific meaning and behavior. Therefore, relationships serve as an important organizing paradigm in such systems. In this article, we address the limitations of the traditional object-oriented model in the area of relationship modeling by providing a core set of semantic relationships, "menus" of choices for additional behavior, and a means for defining new relationship types derived from the core set. This is done in the context of ArchObjects, which serves as an example of how domain-specific relationships may be derived from the core relationship types. Therefore, this approach can be seen as beneficial in the area of architectural modeling. It is highly likely that other complex domains, such as geographic information systems or molecular design systems, could benefit as well from a model that integrates semantic and object-oriented modeling.

Acknowledgments

This research was partially supported by URI Proposal Development Grant 537116. The work was done while Bonnie MacKellar was at the Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, and while Michael Doherty was at the University of Rhode Island, Kingston, RI. The authors would like to acknowledge the help of Zhenghong Dong, Marsha Roberts, and Falguni Vora, who contributed much in the conception and implementation of the SORAC and ArchObjects prototypes.

References

Albano, A., Ghelli, G., and Orsini, R. A relationship mechanism for a strongly typed object-oriented database programming language. *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, 1991.

Andrews, T. and Sinkel, K. Ontos: A persistent database for C++. In: Gupta, R. and Horowitz, E., eds., *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD.* Prentice Hall, Englewood Cliffs, NJ, 1991, pp. 387-406.

Barsalou, T., Keller, A., Siambela, N., and Wiederhold, G. Updating relational databases through object-based views. *Proceedings of the 1991 ACM SIGMOD,* Denver, CO, 1991.

Batory, D. and Kim, W. Modeling concepts for VLSI CAD objects. *ACM TODS,* 10(3):322-346, 1985.

Bouzeghoub, M. and Metais, E. Semantic modeling of object oriented databases. *Proceedings of the Seventeenth International Conference on Very Large Data Bases,* Barcelona, Spain, 1991.

Ceri, S. and Widom, J. Deriving production rules for constraint maintenance. *Proceedings of the Sixteenth VLDB Conference,* Brisbane, Australia, 1990.

Chen, P. The entity-relationship mode: Towards a unified view of data. *TODS,* 1(1):9-36, 1976.

Diaz, O. and Gray, P.M.D. Semantic-rich user-defined relationships as a main constructor. In: Meersman, R., Kent, W., and Khosla, S., eds., *Object Oriented Databases: Analysis, Design, and Construction (DS4).* New York: North-Holland, 1991, p. 207-224.

Dogac, A., Ozkarahan, E., and Chen, P. An integrity system for a relational database architecture. Technical Report 86-03, Department of Computer Engineering, Middle East Technical University, 06531, Ankara, Turkey, 1986.

Doherty, M., Peckham, J., and Wolfe, V.F. Implementing relationships and constraints in an object-oriented database using monitors. *Proceedings of the First International Conference on Rules in Database Systems,* Edinburgh, Scotland, 1993. In: *Workshops in Computing Series,* Springer-Verlag, 1993.

Dong, Z. Design of a user interface for database schema design and analysis. M.S. Thesis proposal, Computer Science and Statistics, The University of Rhode Island, January 1992.

Dube, T. and MacKellar, B. Modeling 3-D building designs with semantic relationships. *Third Eurographics Workshop on Object-Oriented Graphics,* Champery, Switzerland, 1992.

Eastman, C., Bond, A., and Chase, S. A data model for design databases. *Proceedings of the First International Conference on Artificial Intelligence and Design,* Edinburgh, Scotland, 1991.

Foo, S. and Takefuji, Y. Databases and cell selection algorithms for VLSI cell libraries. *IEEE Computer,* 23(2):18-30, 1990.

Gehani, N. and Jagadish, H.V. Ode as an active database: Constraints and triggers. *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, 1991.

Geller, J. Propositional representation for graphical knowledge. *International Journal of Man-Machine Studies,* 34:97-131, 1991.

Halper, M., Geller, J., and Perl, Y. *Part* relations for object-oriented databases. *Proceedings of the Eleventh International Entity-Relationship Conference,* Karlsruhe, Germany, 1992.

Hammer, M. and McLeod, D. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems,* 6(3):351-386, 1981.

Hudson, S.E. and King, T. The Cactis project: Database support for software environments. *IEEE Transactions on Software Engineering,* 14(6):00-00?, 1988.

Hull, R. and King, R. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys,* 19(3):201-260, 1987.

Kim, W., Bertino, E., and Garza, J.F. Composite objects revisited. *Proceedings of the ACM SIGMOD,* Portland, OR, 1989.

Kim, W. *Introduction to Object-Oriented Databases,* Computer Systems Series, Cambridge, MA: MIT Press, 1990.

Kim, W. and Lochovsky, F., eds., *Object-Oriented Concepts, Databases, and Applications.* New York: Frontier Series, ACM Press, 1989.

Laffra, C. and van den Bos, J. Propagators and concurrent constraints. *OOPS Messenger,* 2:68-72, 1991.

MacKellar, B. and Ozel, F. ArchObjects: Design codes as constraints in an object-oriented KBMS. *Artificial Intelligence in Design,* Oxford: Butterworth/Heinemann, 1991, p. 115-134.

MacKellar, B. A constraint-based model of design object versions. *Proceedings of the Third International Conference on Data and Knowledge Systems for Manufacturing and Engineering,* Lyon, France, 1992.

MacKellar, B. and Peckham, J. Representing design objects in SORAC: A data model with semantic objects, relationships and constraints. In: Gero, J.S., ed. *Artificial Intelligence in Design,* The Netherlands: Academic Publishers, 1992, pp. 201-219.

Morgenstern, M. Constraint equations: Declarative expression of constraints with automatic enforcement. *Proceedings of the Tenth International Conference on Very Large Databases,* Singapore, 1984.

Nguyen, G.T. and Rieu, D. Representing design objects. *Artificial Intelligence in Design,* Oxford: Butterworth/Heinemann, 1991, pp. 367-386.

ONTOS Developer's Guide. Burlington, MA: Ontologic, Inc., 1991.

Peckham, J. and Maryanski, F. Semantic data models. *ACM Computing Surveys,* 20(3):153-189, 1988.

Peckham, J., Maryanski, F., Beshers, G., Chapman, H., and Demurjian, S.A. Constraint-based analysis of database update propagation. *Proceedings of the Tenth International Conference on Information Systems,* Boston, MA, 1989.

Peckham, J., Maryanski, F., and Demurjian, S. Towards the correctness and consistency of update semantics in semantic database schema. *IEEE Transactions on Knowledge and Data Engineering,* 1995, to appear.

Prichard, J., Peckham, J., Cingiser, L., and Wolfe, V.F. RTSORAC: Design of a real-time object-oriented database system. *Fifth International Conference on Database and Expert Systems Applications,* Athens, 1994.

Qian, X. A constraint-based database schema checking system. M.S. Thesis, The University of Rhode Island, Kingston, RI, 1994.

Roberts, M. C++ Implementation of ArchObjects Symbolic Framework. Department of Computer and Information Science, New Jersey Institute of Technology, 1993.

Rumbaugh, J. Relations as semantic, constructs in object-oriented language. *ACM OOPSLA Proceedings,* 1987.

Sathi, A., Fox, M.S., and Greenberg, M. Representation of activity knowledge for project management. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 5:00-00?, 1985.

Stefik, M. and Bobrow, D.G. Object-oriented programming: Themes and variations. *AI Magazine,* 6(4):40-62, 1986.

Urban, S. ALICE : An assertion language for integrity constraint expression. *Proceedings of COMPSAC '89,* Orlando, FL, 1989.

Urban, S. and Delcambre, L.M. Constraint analysis : Identifying design alternatives for operations on complex objects. *Transactions on Knowledge and Data Engineering,* 2(4):391-400, 1990.

Urban, S., Karadimce, A., and Nannapaneni, R. The implementation and evaluation of integrity maintenance rules in an object-oriented database. *Proceedings of the Eighth IEEE Data Engineering Conference,* Tempe, AZ, 1992.

Vora, F. ARAC: A data modeling interface for architectural design systems. M.S. Thesis, Computer Science and Statistics, The University of Rhode Island, 1992.

Widom, J. and Finkelstein, S.J. Set-oriented production rules in relational database systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Atlantic City, NJ, 1990.

Zdonik, S. Fundamentals of object-oriented databases, introduction. In: Zdonik, S. and Maier, D., eds., *Readings in Object-Oriented Database Systems,* San Mateo, CA: Morgan Kaufman, 1990, pp. 1-36.