Management of Complex Structural Engineering Objects in a Relational Framework

Kincho H. Law, Thierry Barsalou and Gio Wiederhold

TECHNICAL REPORT
Number 19

September, 1989

Stanford University
Copyright © 1989 by
Center for Integrated Facility Engineering

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020
Management of Complex Structural Engineering Objects in a Relational Framework*

Kincho H. Law‡, Thierry Barsalou‡ and Gio Wiederhold§

February 12, 1990

ABSTRACT

To structure the development of an integrated building design environment, the global representation of the design data may best be organized in terms of hierarchies of objects. In structural engineering design, we deal with large sets of independent but interrelated objects. These objects are specified by data. For an engineering design database, the system must be able not only to manage effectively the design data, but also to model the objects composing the design. The database management system therefore needs to have some knowledge of the intended use of the data, and must provide an abstraction mechanism to represent and manipulate objects. Much recent research in engineering databases focuses on object management for specific tasks but gives little attention to the sharability of the underlying information. This paper describes an architecture for the management of complex engineering objects in a sharable, relational framework. Potential application of this approach to object management for structural engineering analysis and design is discussed.

*An earlier version of this paper has been accepted for publication in the Journal of Engineering with Computers

‡Department of Civil Engineering, Stanford University, Stanford, CA 94305
§Section on Medical Informatics, Stanford University, Stanford, CA 94305
¶Department of Computer Science, Stanford University, Stanford, CA 94305
1 Introduction

In building design, we deal with large sets of independent but interrelated objects. These objects are specified by data. The data items describe the physical components (for example, columns, beams, slabs) and the topological aspects of the design (for example, member and joint connectivities). The design data need to be stored, retrieved, manipulated, and updated, during all phases of analysis, design, and construction of the project. An efficient data management system becomes an indispensable tool for an effective integrated computer aided analysis and design system.

Using a database to store and describe engineering data offers many benefits [29]. Some of these benefits include:

- Ability to store and access data independent of its use, so that the data can be shared among the participants
- Ability to represent relationships among the data, so that dependencies are documented
- Control of data redundancy, so that consistency is enhanced
- Management of data consistency and integrity, so that multiple users can access information simultaneously
- Enhanced development of application software by separating data management function
- Support of file manipulation and report generation for ad hoc inquiry

To maximize these benefits, a database management system needs to have some knowledge of the intended use of the data. That is, the formal structure or model used for organizing the data must be capable of depicting the relationships among the data and must facilitate the maintenance of these relationships. Furthermore, the structure should be sufficiently flexible to allow a variety of design sequences and to aid an engineer to understand the design.

Traditional relational database systems provide many interesting features for managing data; among them are the capabilities of set-oriented access, query optimization and declarative languages. More important, from the user point of view, the relational model is completely independent of how data are physically organized. The relational model presents data items as records (tuples) which are organized in 2-dimensional tables (relations), and provides manipulation languages (relational calculus and algebra) to combine and reorganize the tables or relations for processing. The relational approach is simple and effective, particularly for business data processing. However, a “semantic” gap exists between the relational data model and engineering design applications. The relationships among the data items describing an engineering design are often complex. The lack of a layered abstraction mechanism in the relational model makes it inadequate for defining the semantics of applications and for maintaining the interdependencies of related data items. Furthermore, the traditional
set-oriented relational structure does not support well the engineering views of the data. The engineering users have to supply all the intentional semantics in order to exploit the data.

Object-orientation is an active focus of engineering database research. Object-oriented data models have been proposed to increase the modeling capability, to provide richer expressive concepts and to incorporate some semantics about engineering data. The main objective here is to reduce the semantic gap between complex engineering design process and the data storage supporting the process. In such a process, an engineer often approaches the design in terms of the components (objects) that comprise the design, and the operations (methods) that manipulate the components. A database system that supports the object-oriented nature of the design process can certainly enhance the interactions between the engineers and the system.

It should be noted that an object-oriented data model does not necessarily imply that the object-oriented paradigm need to be explicitly implemented inside the database system. In engineering modeling and design, the information that an object represents is often shared by various applications having different views of the data. Data sharing is therefore as important as object-oriented access. Storing objects (explicitly) in object format is not desirable, particularly if the objects are to be shared [31]. We therefore propose an approach, based on the structural data model, that permits object-oriented access to information stored in a relational database; information which in turn can be shared among different applications [2, 5, 6, 24]. In this paper, we discuss the application of this model for the management of complex structural engineering objects in a relational framework.

This paper is organized as follows: The needs of a structural engineering database system are discussed in Section 2. The structural data model is briefly reviewed in Section 3. In Section 4, we present the principles and motivation of an object-oriented system (PENGUIN), its architecture, and some structural engineering examples. In Section 5, we discuss the use of view-objects for modeling design abstractions. In Section 6, we conclude this paper with a summary of the expected benefits of our approach for engineering design applications.

2 Abstractions in Structural Engineering

Practical engineering tasks have too many relevant facets to be intellectually represented through a single abstraction process. Manageability of an application can be achieved by decomposing the model into several hierarchies of abstractions. In general, an aspect of a building and its design can be described as a collection of objects or concepts organized hierarchically [12, 14, 15, 18, 22, 25, 27]. The description of a design project grows as it evolves. During the design, additional attributes may be added to the description of existing entities; similarly, aggregated entities can be decomposed into their constituents. That is, during design, information is added to the hierarchy by refinement in a top-down manner or by aggregation in a bottom-up sequence. The concept of abstraction provides a means for defining complex structures as well as the semantic information about the data. Powell and Bhateja have defined
some requirements for an abstraction model in structural engineering application [25] :

- The model must support several applications.
- The model should be in terms of well-defined entities, relationships and dependencies.
- The model must support the creation of abstractions for real structures; that is, it must allow all relevant features of a structure to be represented. In addition, the concepts used in the model should be familiar to the users.
- The model should allow the level of details to be increased as the design of the structure is progressively refined.
- The model should be able to represent structures of various types.

A structural engineering database system must be capable of supporting such an abstraction model.

Choosing a good data model to represent design data and processes is a major step towards the development of an integrated structural design system. A data model is a collection of well-defined concepts that help the database designer to consider and express the static properties (such as objects, attributes and relationships) and the dynamic properties (such as operations and their relationships) of data intensive applications [10]. In addition to enhancing the database design process, a data model must also provide the integrity rules to ensure consistency among the entities.

A relationship is a logical binding between entities. There are three basic types of relationships that are commonly used: association, aggregation and generalization. Association relates two or more independent objects as a merged object, whose function is to represent the many-to-many relationships among the independent objects. Association can be used to describe multiple “member of” relationships between member objects and a merged object. Aggregation combines lower level objects into a higher level composite object. In general, aggregation is useful for modeling part-component hierarchies and representing “part of” relationships. Generalization relates a class of individual objects of similar types to a higher level generic object. The constituent objects are considered specializations of the generic object. Generalization is useful for modeling alternatives and representing “is a” relationships. These three basic relationship types, in particular aggregation and generalization, are supported by many semantic data models and have been widely used in computer aided building design research [8, 14, 15, 19, 21, 22]. A joint can be represented as an association of several structural elements (such as beams and columns) and connecting plates, and carries some information about the connectors to be used. A staircase is an aggregation of many similar parts. A concept “beams” is a generalization of a variety of members supporting gravity loads.

These three relationships impose certain existential dependency among the object entities. For example, the joint information is only meaningful while the referencing beams and plates are part of the design. As another example, assuming that the
entities "BEAM" and "COLUMN" are specializations of a generic entity "STRUCTURAL ELEMENT", existence of a "BEAM" or "COLUMN" instance requires that a corresponding instance also exists in the generic entity "STRUCTURAL ELEMENT". When an instance is deleted from the generic entity "STRUCTURAL ELEMENT", corrective measure should be taken to remove the corresponding instance in a specialized entity "BEAM" or "COLUMN"; as a result, consistency between the generic and the specialized entities can be maintained in the database. Dependency constraints of these three types of relationships have been examined in details [3, 9, 11, 23, 26].

Besides association, aggregation and generalization, other relationships, such as "connected to", "supported by" (or "supporting"), "influence" and "determinants", have received considerable interests in building design applications [1, 13, 18, 25]. While incorporating these types of relationship into a data model is desirable, dependencies among the entities participating in these relationships have not yet been formally defined. For instance, when a supporting element is removed from a structure, corrective measures should be imposed on the objects that it supports. On the other hand, removing a supported element, from the data management point of view, should have little effect on the corresponding supporting object. It is important for a database management system to ensure consistency among the building design data, and to reflect appropriately the semantic relationships among them.

To be general, a database must support all design activities, in addition to capturing the semantic knowledge of the data. For each activity, an engineer works with specific application abstraction of the building, rather than with a complete physical description. For example, in the analysis of a building structure, an engineer is primarily interested in the building frame in terms of the center line of the members, their physical properties and the stiffness of the connections. Other information, such as room spaces and wall partitions, can be ignored. The database system needs to support various abstract views of the information pertaining to a specific domain. The ability to support multiple views for satisfying the requirements of different applications is also an important aspect of an engineering data model.

3 The Structural Data Model

The goal in selecting a semantic data model is to represent directly in an easily manipulable form as many of the objects and their relationships of interest as possible. The structural data model that we use in this study is an extension of the relational model [16, 30]. Relations are used to capture the data about objects and their parts. The structural data model augments the relational model by capturing the knowledge about the constraints and dependencies among the relations in the database. This section reviews briefly the structural data model. For more detailed description of this data model, the reader is referred to References [3, 5, 16, 30].

The primitives of the structural data model are the relations and the connections formalizing relationships among the relations. The connection between two relations $R_1$ and $R_2$ is defined over a subset of their attributes $X_1$ and $X_2$ with common domains. Two tuples, $t_1 \in R_1$ and $t_2 \in R_2$, are connected if and only if the connecting
attributes in \( t_1 \) and \( t_2 \) match. There are three basic types of connections, namely \textit{ownership}, \textit{reference}, and \textit{subset} connections. These connections are used to define the relationships between the relations and to specify the dependency constraints between them.

![Diagram of building structure]

Figure 1: Composition of a Building Structure using the Structural Model

An \textit{ownership} connection between an owner relation \( R_1 \) and an owned relation \( R_2 \) is useful for representing "part-component" or aggregation type of relationship. As an example, Figure 1 shows the composition of a building structure consisting of a few simple entities. The basic components of a building structure include the descriptions of structural elements, floor, foundation, roof, space, etc.. The components exist if and only if the building exists. This owner-component relationship is best represented using the ownership connection. This connection type specifies the following constraints:

1. Every tuple in \( R_2 \) must be connected to an owning tuple in \( R_1 \).

2. Deletion of an owning tuple in \( R_1 \) requires deletion of all tuples connected to that tuple in \( R_2 \).
The ownership connection describes the dependency of multiple owned tuples on a single owner tuple.

A reference connection between a primary (referencing) relation $R_1$ and a foreign (referenced) relation $R_2$ is useful for representing the notion of abstraction. Referring to the example shown in Figure 1, an architectural space, which may be an office, elevator opening, mechanical room etc., locates on (or references) a floor. The floor cannot be removed without first removing the spaces defined on that floor. This connection type specifies the following constraints:

1. Every tuple in $R_1$ must either be connected to a referenced tuple in $R_2$ or have null values for its attributes $X_1$.

2. Deletion of a tuple in $R_2$ requires either deletion of its referencing tuples in $R_1$, assignment of null values to attributes $X_1$ of all the referencing tuples in $R_1$, or assignment of new valid values to attributes $X_1$ of all referencing tuples corresponding to an existing tuple in $R_2$.

The reference connection describes the dependency of multiple primary tuples on the same foreign tuple. The reference connection can be used to refer to concepts which further describe a set of related entities. It should be noted that association can be modeled with a combination of ownership and reference connections [32]. As an example, in a steel frame structure, a joint connection is associated with the structural elements through one or more connectors (see Figure 3).

A subset connection between a general relation $R_1$ and a subset relation $R_2$ is useful for representing alternatives or “is a” type relationship. Generalization (and its inverse, specialization) can be modeled using the subset connection. For the example shown in Figure 1, a structural element can either be a column, a beam, a wall, or a slab. Furthermore, a beam can be generalized to be either a main girder or a joist (secondary beam). Deleting a specific instance in the generic class of structural element must delete the corresponding instance existing in the subclass. The subset connection specifies the following constraints:

1. Every tuple in $R_2$ must be connected to one tuple in $R_1$.

2. Deletion of a tuple in $R_1$ requires deletion of the connected tuple in $R_2$.

The subset connection links general classes to their subclasses and describes the dependency of a single tuple in a subset on a single general tuple.

Besides supporting the three basic relationships of aggregation, generalization and association, the connections can also be used to define relationships such as “connected-to” and “supported-by”, that are useful in engineering application. In the “supported-by” or “supporting” relationship, the supporting entity should not be removed unless all its supported entities no longer exist. For example, when a wall is removed from a design, the openings, such as windows and doors, located inside that wall have to be removed also. As another example, a column should not be removed unless the references from the components such as walls, beams, slabs that
the column supports no longer exist. This dependency property can be modeled using
the reference connection as shown in Figure 2.

One application of the "connected-to" relationship in structural engineering is for
the description of a joint connecting structural elements. As an example, we can
represent a joint connection that is described in an interactive modeling system for
the design of steel framed structure (Steelcad) [28]: "The connected members and
the joint are logically linked in the database:

- If a joint is removed, then the previously connected members are automatically
  restored, as they were before the connection was defined.

- If a member is removed, then all connections that it has with other members
  are also removed and the other members reappear accordingly."

As shown in Figure 3, this description of a joint can be modeled using the three basic
connection types. We assume that a joint connection may consist of one or more
connectors. Each connector references a structural member and a connecting element.
That is, deleting a connector does not affect the connecting members. On the other
hand, if a structural member or a connecting plate is removed, the connectors are
also removed.

4 An Object Management System in a Relational Framework

In the previous section, we have briefly discussed the structural data model and
its three formal connection types. We have shown that the model can be used to
represent various relationships that are useful in structural engineering applications. In this section, we describe an architecture, based on the structural data model, for the management of complex objects in a relational framework and a prototype implementation of that approach in the PENGUIN system. In addition to its ability to capture the semantic relationships among the entities, PENGUIN can handle multiple views of design data and multiple representations of design objects.

4.1 General Principles

The object-oriented paradigm has gained much attention in computer aided design in recent years; it offers many advantages over traditional relational data model. Object-oriented systems help managing related data having complex structure by combining them into objects. The use of objects permits the user to manipulate the data at a higher level of abstraction. However, storing objects poses a problem when these objects are to be shared by multiple engineering design tasks. The amount of information pertaining to an object grow as each design task requires different information about the object. As a design progresses, an object may become too complex to be efficiently managed. The benefits of understandability and naturalness of having objects are lost [31]. Besides the problem of object sharing, object-oriented systems do not provide those indispensable features of DBMSs such as file management structures and concurrency control. Processing of queries involving large and complex sets of data is also not well supported by object-oriented systems.

Another approach is to store the objects explicitly in a relational database. In engineering application, we deal with entities that are more complex than single tuples or sets of homogeneous tuples. Quite frequently, an object is a hierarchical group of tuples comprising of a single root tuple that defines the object, and one or more dependent tuples that further describe the object’s properties. Because of normalization theory, these dependent tuples reside in one or more relations distinct from the relation containing the root tuple. Even if such a structure is easily expressed relationally (through joins), it cannot be manipulated as a single entity. We need to
make such structures explicitly known to the system. Furthermore, as noted earlier, different users require different views of the information included in an object. Update anomalies and problems of redundancy would arise if the objects corresponding to the different views were to be stored as such lattice. Last but not least, changes to the set of classes and to the inheritance can be made quite frequently at various stages of a design project. If the objects were explicitly stored, the schema would have to be changed accordingly.

Our approach is therefore to define and manipulate complex objects that are constructed from base relations. Our prototype implementation, PENGUIN, keeps a relational database system as its underlying data repository. Indeed, we believe that the relational model should be extended rather than replaced. The relational model has become a de facto standard and thus some degree of upward compatibility should be kept between the relational format and any next-generation data model. We augment the relational model with the structural data model. The PENGUIN system uses the structural model together with the traditional data schema to define the object schema. The idea is not to store the objects directly in persistent form but rather to store their description, which are used later to instantiate objects as needed.

4.2 Architecture

Both the concepts of view and object are intended to provide a better level of abstraction, bringing together related elements, which may be of different types, into one unit. For example, to display a building frame, we bring together structural entities, such as beams, columns and connections. The architecture for combining the concepts of views and objects has been initially proposed by Wiederhold [31]. A set of base relations serve as the persistent database and contain all the data needed to create any specific view or object. We then extract the data corresponding to a view-object from a relational database system and assemble the data into object instances based on the definition of the object template specified in terms of the connections of the structural data model [2, 5, 6, 7]. Multiple layers of view-objects can be defined so that a view-object can be expressed in terms of other view-objects and can be shared by other view-objects. Figure 4 summarized this notion of multiple object layers and their interaction with the underlying set of relations.

A prototype system (PENGUIN) for this object-based architecture is being implemented by Barsalou [4]. The schematic diagram of the PENGUIN system architecture is shown in Figure 5. The system architecture consists of three basic components: an object (template) generator, an object instantiator and an object decomposer. Each view-object is defined by an object template. The object generator maps relations into object templates where each template can invoke join (combining two relations through shared attributes) and projection (restricting the set of attributes of a relation) operations on the base relations. To define an object template, we first select a pivot relation from a set of base relations. The key of the pivot relation corresponds to the primary object key. Further data is related to this object by following the connections of the structural data model. By organizing an object template around a
pivot relation, each object instance can be uniquely identified by the value of the key of its pivot relation. The relations that are connected to the pivot relation through structural connections become potential candidate relations to be included in the object template. Once the pivot relation is specified, PENGUIN automatically derives a set of candidate relations from the connections of the structural data model. Secondary relations and their attributes (including those of the pivot relation) can then be selected from the set of candidate relations. Once an object template is defined, data access functions are derived to facilitate the data retrieval process. Related templates can be grouped together to form an object network, identifying a specific object view of the relational database. The whole process is knowledge-driven, using the semantics of the database structure as defined by the connections of the structural data model.

The object instantiator provides nonprocedural access to the actual object instances. The instantiator performs all the operations for information retrieval and manipulation that are necessary to instantiate and display an object template. A declarative query specifies the template of interest. Combining the database-access function and the specific selection criteria, the system automatically generates the relational query and transmits it to the relational database system, which in turn transmits back the set of matching relational tuples. The retrieved tuples are assem-
Figure 5: PENGUIN's Architecture: An Object Layer on Top of a Relational DBMS

bled into the desired object instances, based on the semantics defined in the object template.

The object decomposer maps the object instances back to the base relations. This component is invoked when changes to some object instances need to be made persistent at the database level. An object is generated by collapsing (potentially) many tuples from several relations. Similarly, one update operation on an object may result in a number of update operations on several base relations. Dependency constraints are enforced to ensure the database consistency. These actions are based on the integrity rules imposed by the connections of the structural data model. Since the object templates are defined using join operations on the database relations, we are then facing the well-known problem of updating relational databases through views involving multiple relations [17]. Updating through views is inherently ambiguous, as a change made to the view can translate into different modifications to the underlying database. Keller has shown that, using the structural semantics of the database, one can enumerate such ambiguities, and that one can choose a specific translator at view-definition time [20]. Because of the analogy between relational views and PENGUIN's object templates, Keller's algorithm applies to our approach. When creating a new object template, a simple dialogue, the content of which depends on the structure of the object template, allows the user to select one of the semantically valid translators. The chosen translator is then stored as part of the template definition. When an object instance is modified, the object decomposer will use this information to resolve any ambiguity completely and to update the database correctly [5].

12
4.3 An Example

![Diagram of a simple frame structure with relations]

**Relation Member**

<table>
<thead>
<tr>
<th>Member_ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM.1</td>
<td>Beam</td>
</tr>
<tr>
<td>BM.2</td>
<td>Beam</td>
</tr>
<tr>
<td>BM.3</td>
<td>Beam</td>
</tr>
<tr>
<td>COL.1</td>
<td>Column</td>
</tr>
<tr>
<td>COL.2</td>
<td>Column</td>
</tr>
<tr>
<td>COL.3</td>
<td>Column</td>
</tr>
<tr>
<td>COL.4</td>
<td>Column</td>
</tr>
<tr>
<td>COL.5</td>
<td>Column</td>
</tr>
<tr>
<td>COL.6</td>
<td>Column</td>
</tr>
</tbody>
</table>

**Relation Beam**

<table>
<thead>
<tr>
<th>Beam_ID</th>
<th>Design</th>
<th>N_ID1</th>
<th>N_ID2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM.1</td>
<td>W24X76</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>BM.2</td>
<td>W24X68</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>BM.3</td>
<td>W24X68</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**Relation Column**

<table>
<thead>
<tr>
<th>Column_ID</th>
<th>Design</th>
<th>N_ID1</th>
<th>N_ID2</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL.1</td>
<td>W14X82</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>COL.2</td>
<td>W14X82</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>COL.3</td>
<td>W14X61</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>COL.4</td>
<td>W14X61</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>COL.5</td>
<td>W14X53</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>COL.6</td>
<td>W14X53</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

**Relation Node**

<table>
<thead>
<tr>
<th>NODE_ID</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>30.0</td>
</tr>
<tr>
<td>2</td>
<td>30.0</td>
<td>0.0</td>
<td>30.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>12.0</td>
<td>30.0</td>
</tr>
<tr>
<td>4</td>
<td>30.0</td>
<td>12.0</td>
<td>30.0</td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
<td>24.0</td>
<td>30.0</td>
</tr>
<tr>
<td>6</td>
<td>30.0</td>
<td>24.0</td>
<td>30.0</td>
</tr>
<tr>
<td>7</td>
<td>0.0</td>
<td>36.0</td>
<td>30.0</td>
</tr>
<tr>
<td>8</td>
<td>30.0</td>
<td>36.0</td>
<td>30.0</td>
</tr>
</tbody>
</table>

**Relation Geom_PROP**

<table>
<thead>
<tr>
<th>Design</th>
<th>S</th>
<th>A</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>W24X76</td>
<td>176</td>
<td>22.4</td>
<td>2100</td>
</tr>
<tr>
<td>W24X68</td>
<td>154</td>
<td>20.1</td>
<td>1830</td>
</tr>
<tr>
<td>W14X53</td>
<td>77.8</td>
<td>15.6</td>
<td>541</td>
</tr>
<tr>
<td>W14X61</td>
<td>92.2</td>
<td>17.9</td>
<td>640</td>
</tr>
<tr>
<td>W14X82</td>
<td>12.3</td>
<td>24.1</td>
<td>882</td>
</tr>
</tbody>
</table>

Figure 6: A Set of Base Relations for a Simple Frame Structure

We now illustrate the construction of an object network using the PENGUIN system with a simple structural engineering example. Figure 6 shows a set of base relations and their connections that describe a simple frame structure. As shown in Figure 7, the relations and the connections are specified using PENGUIN's graphical interface. To create a complex "MEMBER OBJECT", we define a template organized around the pivot relation "MEMBER" in the underlying database. Each instance of the "MEMBER OBJECT" is thus uniquely identified by the key attribute values of the pivot relation "MEMBER". Once the pivot relation has been specified, a candidate
graph containing the valid relations is derived and is converted into a candidate tree where the root is the pivot relation and all other nodes are secondary relations that can be included in the object template. The candidate relations for the "MEMBER OBJECT" template are shown in Figure 8.

![Diagram showing the structural model of a frame with nodes and relations]

**Figure 7: Defining an Object Template Using PENGUIN’s Graphical Interface**

Once the candidate tree is established, the user can specify any number of secondary relations and their attributes to be included in the object template. For example, we can include the information about beam members as shown in Figure 9 and define it as "BEAM OBJECT". Based on this information, the system automatically derives the database access function, the linkage of the various data elements within the object, and the compulsory attributes that are required for performing join operations on the selected relations. Such a view-object can now be exploited by engineering applications. For example, the query (retrieve BEAM-OBJECT with Member-Id = ‘BM-1’) would fetch the view object instance displayed in Figure 10.
Figure 8: The Candidate Tree of an Object Template Generated by PENGUIN

Figure 9: Definition of a View-Object for Beams

Now, let us assume that a substructure-component definition and the corresponding relations have been entered as shown in Figure 11. We can then create an object template "SUBSTRUCTURE OBJECT" with the pivot relation "SUBSTRUCTURE". As shown in Figure 12, this newly created object template can be inserted into an object network by simply connecting it to other object templates previously defined (thereby updating the object schema.) Since the object-network connections are abstracted from the underlying database structural connections, the relationships are explicitly carried over to the object layer and can be used to provide inheritance of attributes among the templates.

5 View-Objects and Design Abstractions

As the design process progresses from conceptualization to design, the way to represent the design is constantly changing. For an integrated design system to be effective, the database system must be able to accommodate the "growth" of the design. As mentioned earlier, the data model must support a wide variety of design representations, sharing the same information in the engineering model. In addition, the data model must allow dynamic changes of the object schema, reflecting the evolutionary process of design, and must be able to minimize database reorganization. The view-object facility described in the previous section allows the engineer to select the object information pertinent to a design task and to ignore the irrelevant details. Furthermore, the separation of the object schema and the database schema can facilitate schema
Figure 10: An Instance of the View-Object BEAM-OBJECT

evolution during the design process.

An abstraction view of a building and its components is not unique. An engineer abstracts a specific view of design to focus on a particular task. As an example, an hierarchical decomposition of a building structure is shown in Figure 13(a). As shown in Figure 13(b), for design purposes, a floor composed of beams (including girders and joists), slabs, columns can be conveniently treated as objects. For analysis purposes, a building frame composed of girders and columns is defined as shown in Figure 13(c). We see here two multiple design views sharing the same information. For the respective views, the attributes of a shared entity are not the same. For the description of a floor plan, only the location and orientation of the columns are important. For frame analysis, however, the location, the dimension and the properties of the columns are needed.

By allowing the user to select any number of secondary relations for inclusion, an object can be specified to any level of details that is desired. For example, in the earlier analysis stage, a floor may be treated as the basic component entity but may be expanded to include other subcomponents such as beams and slabs at a later stage of the design; an object schema can be changed dynamically as the design evolves. Hence, the complexity of a design can be managed by suppressing the irrelevant details as necessary. Furthermore, the description of an entity can be refined as needed.

In Figure 13(c), the entity “FRAME” is composed of subentities “GIRDER” and “COLUMN”. However, removing a “FRAME” instance does not necessarily imply that all its constituent components be deleted as well. As shown in Figure 13(d), an alternative may be to augment the “FRAME” object with the auxiliary relations “FRAME GIRDER” and “FRAME COLUMN”, which reference relations “GRIDER” and “COLUMN”, respectively, and are owned by relation “FRAME”. This new structure provides an associative relationship such that removing a “FRAME” instance does not affect the base relations “COLUMN” and “GIRDER”; however, a column cannot be deleted as long as a structural frame containing that column exists. A similar view can be created for the abstract entity “FLOOR”. It should be empha-
sized that modifications made to an individual object template does not necessarily lead to modification of a higher level object. Conversely, modifications of the object network do not affect the definition of the base relations.

6 Summary and Discussion

In this paper, we have examined the potential applications of the structural data model in structural engineering. The connections of the structural data model properly define the various relationships, and their constraints and dependencies that are of interest to researchers in computer aided building design. Besides being an effective database design tool, the structural data model can serve as the basis for the development of an object interface to a relational database system, supporting multiple object views. The architecture of an object management system has also been briefly described. The application of this object model for structural analysis and design has been discussed.

Many advantages of this view-object approach can be identified:

- It provides multiple views of the stored information that is relevant to an engineering model or design
Figure 12: Inserting a New Object Template into an Object Network Using PENGUIN

- It provides a mechanism for storing objects, independent of one specific view or application
- It eliminates the data redundancy since the information about an object is shared but not duplicated
- It maintains integrity of the data for a given object based on the structural constraints that are specified by the connections
- It supports growth of a design because of the logical independence between the object definitions and the database schema.

The key benefit of the view-object interface to a relational system, besides information sharing, is that any new attributes and/or relations added to the underlying database do not affect the object definitions. Conversely, changes in the definition of any objects do not affect the schema of the underlying database. In other words, the architecture is sufficiently flexible to allow growth as design evolves.
7 Acknowledgment

The work by Dr. Thierry Barsalou and Professor Gio Wiederhold is supported by the National Library of Medicine under Grant R01-LM04836, DARPA under Contract N39-84-C-0211 for Knowledge Base Management System, and Digital Equipment Corporation under the Quantum project. The work by Professor Kincho H. Law is supported by the Center for Integrated Facility Engineering at Stanford University.

References


