ODEPSI: An Experimental Object-Oriented Data Base Management System

JAE-JUN KIM AND C. WILLIAM IBBS

ODEPSI is an experimental object-oriented data base management system. Developed as an extension to Smalltalk-80™, ODEPSI is currently designed for the management of design and construction project data. ODEPSI represents and manages data items on the basis of the object-oriented paradigm; all data items are in objects, and each object has a set of properties and related procedures. The current version of the system emphasizes automatic constraint enforcement and semantic modeling. ODEPSI provides facilities for defining and maintaining constraints on property values, object types, and object relationships. Semantics, which imply the meaning of the real-world subject being modeled, are represented through the use of object properties and interobject relationships. Major system components of the current version include a data definition language and a data manipulation language.

Data base technology is an ideal tool for integrating various design and construction process functions (*1–3*). Its ability to store, retrieve, sort, analyze, update, delete, and trace data of various kinds, which are generated through all phases of the building cycle, provides a crucial foundation for integration.

A data base is developed with a certain structure that defines the relationships among its entities or record types (*4*). This structure is called a data model. Notable data models in use are the network, hierarchical, and relational forms. Software that implements one of these data models is called a data base management system (DBMS) and is used for the creation, manipulation, and maintenance of a data base.

The relational data base management system (RDBMS), which is based on the relational data model, is the most favored DBMS in construction today. The RDBMS gained popularity over other data base systems for its simplicity and ease of use. Many construction organizations now use data bases as integral parts of their project control systems.

The problem with the relational data model is that it lacks semantic expressiveness, which is an ability to convey the meaning of a real-world subject through the represented data elements. Dynamics of construction projects are not properly depicted in current relational data bases.

The object-oriented data model is an alternative to the traditional pure relational data model for managing construction project data. An experimental object-oriented data base management system (OODBMS) called ODEPSI has been developed. ODEPSI is an acronym for Object-oriented data base management system for DEsign and Project Planning and Control System Integration. The benefit of this approach to project and construction managers is a homogeneous environment for programming and data base that allows better representation of the semantics involved in their projects.

Department of Civil Engineering, University of California, Berkeley, Calif. 94720.

OBJECT-ORIENTED DATA MODELING

Object-oriented data bases are a data modeling and manipulation concept that is based on the concept of an object. Although this concept originated in the 1960s with the SIMULA effort (*5*), it was not fully appreciated until several years later, during the Smalltalk research project at Xerox Palo Alto Research Center. The principal conceptual and functional elements of an object-oriented data base that distinguish it from other data base types are objects, data abstraction, and inheritance.

All data items in an object-oriented data base are treated as objects. Each object is uniquely identified by the system. Any real-world entity, either conceptual or physical, can be represented as an object. Each object can carry and determine its own behavior. This concept of object is an important one, and its benefits will be explained using construction project data.

The existence of an object suggests an interesting comparison from the modeling perspective. In any conventional data base approach, a typical construction project may represent the concept of Activity with a set of data items. Application programs then manipulate this data set to get a meaning of Activity. Because this manipulation process is varied by or within the application programs, careful coordination between the data base and the application programs is needed. However, in an object-oriented data base environment, this external coordination problem no longer exists. The conceptual entity Activity is explicitly represented as an object and its operational behaviors, such as calculating its progress or determining float time, are included within itself.

Other important modeling advantages of an object-oriented data base are its data abstraction and inheritance capabilities. These capabilities are important because they bring efficiency and integrity into the data representation task. Consider a data base storing a structural configuration of a building. Scores of walls may be included. These walls may have instances of two types—exterior and interior. Each wall may have various properties to be described, such as identification, size, material, schedule, cost, and location. One efficient way of organizing these wall data is to make an abstract data object called Wall and to use that object to describe those characteristics common to all walls. Then, this Wall object can be further specialized to differentiate exterior and interior walls—ExteriorWall and InteriorWall. Properties, including behav-

ior, common to these two wall types are defined in the Wall object and inherited to its specializations. Only the necessary characteristics of the two wall types are carried to the lower level.

From this example, the two important concepts of interobject relationships—generalization and specialization—can be observed. Through the data abstraction process of producing a Wall object, a generalization relationship is established. Through the process of subdividing the object into ExteriorWall and InteriorWall objects, a specialization relationship is established. As indicated from this example, one is the mirror image of the other.

In general, the semantics of a real-world subject includes two more concepts of interobject relationships—aggregation and association. In current object-oriented data base environments, these relationships are not explicitly represented. However, using the generalization and specialization concepts as a foundation for object classification, the other relations between objects, aggregation and association, can be represented and maintained.

The importance of this point is indicated by Figure 1, which shows several semantic relationships between building objects. Floor-1 contains three Room instances, Room-1, Room-2, and Room-3. Corridor-1 is part of Floor-1 and adjacent to the three room instances. The three semantic relationships in this example are "contains," "isPartOf," and "isAdjacentTo." The first two relationships, contains and isPartOf, show an example of aggregation; Floor-1 aggregates Room-1, Room-2, Room-3, and Corridor-1. Another relationship, isAdjacentTo, shows an example of association; Corridor-1 is associated with Room-1, Room-2, and Room-3. The following is a list of interobject relationships in the example:

- Floor-1 contains Room-1,
- Floor-1 contains Room-2,
- Floor-1 contains Room-3,
- Floor-1 contains Corridor-1,
- Room-1 isPartOf Floor-1,
- Room-2 isPartOf Floor-1,
- Room-3 isPartOf Floor-1,
- Corridor-1 isPartOf Floor-1,
- Corridor-1 isAdjacentTo Room-1,
- Corridor-1 isAdjacentTo Room-2, and
- Corridor-1 isAdjacentTo Room-3.

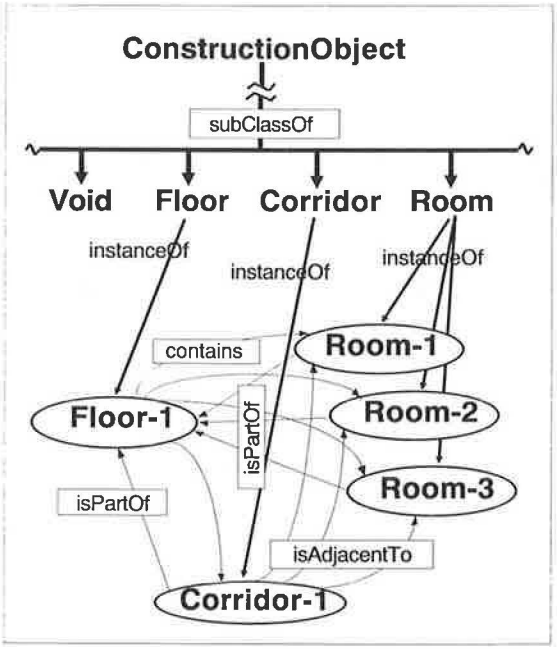

FIGURE 1 Example of aggregation and association.

BASIC ELEMENTS OF OODBMS

The basic elements of an OODBMS, which are commonly recognized within the context of object-oriented paradigm, should be reviewed before ODEPSI is described. The following definitions are based on the Smalltalk-80™–style object-oriented programming (OOP) concept.

Object

An object is a sole data type (class) supported by OODBMS. Other data types may only be defined as a subclass of object. An object-oriented system, therefore, consists of objects of various kinds (including class objects and instance objects) responsible for features that are analogous to an operating system, program, function, or data element of conventional systems. No clear distinction exists for an application program and a data management system. An object is by itself a package of data and related procedures that belong together. In OOP, procedures are sometimes also called methods (*6*).

An object, being a combination of data and methods, allows information hiding and data encapsulation. Information hiding is a concept that simplifies programming tasks by providing minimum components necessary. It is accomplished by surrounding data and methods within the object. Data encapsulation is a concept that increases programming integrity by restricting objects' values accessed or modified only through their methods. This idea in turn allows reduction of the effect of any one software module's change on others. Put another way, information hiding and data encapsulation concepts ensure reliability, modifiability, and safety of software systems by reducing interdependencies between software components (*7–9*).

In OODBMS, each and every object is unique; a hidden, permanent, unique identifier is assigned to each object. This identifier is only visible to system components. If an object is to be queried directly from objects of the same class without comparing their values, an object's identity should be saved explicitly as a global variable. Thus, if a variable is known that references the object to be queried, there is no need to query the object. It is already accessed through the variable that references it.

For this reason, objects can be queried not only by their values but also by their identities. Some researchers of conventional RDBMSs have independently arrived at the importance of objects (*10–12*). They have expanded the traditional value-based approach of the RDBMS by using the object concept of OOP.

Classes and Instances

A class is an object that describes behaviors of similar objects (instances) in terms of class variables, instance variables, and methods. Class definitions are analogous to schemes of traditional DBMS (and instance variables are used similarly to the fields of relational tables), but classes differ from schemes in that they control their own behaviors. Unlike traditional DBMSs, which have separate programs for data manipulation, each OODBMS class is capable of controlling its behavior through the methods encapsulated within its structure. Manipulating data stored in class and instance variables, and performing system-level operations such as constraint enforcement or consistency maintenance, are possible.

Classes are organized hierarchically to facilitate system organization and development. A new class (data type) is created as a subclass of an existing class (which then becomes a superclass) and shares similarities. Methods and instance variables of the superclass are available for the subclass by means of inheritance. A class hierarchy represents a semantic relationship of generalization and specialization. A superclass is a generalization of its subclasses, and a subclass is a specialization of a superclass. These two semantic relationships are complementary to each other and are sometimes described as an is-a relationship.

Instances themselves become objects, each representing a specific case of a class. They are analogous to tuples (records or rows) of relational tables. An instance object's state is captured in its instance variables, and behaviors of an instance are controlled by the methods of its class. To be precise, those methods that control an instance's behaviors are called instance methods. There are other methods, namely class methods, that only respond to classes. Each class has its class methods and typically uses them for the creation of its instances.

Figure 2 shows an example of structural component classes and their instances. The objects in grey bubbles are classes, and the objects in white bubbles are instances.

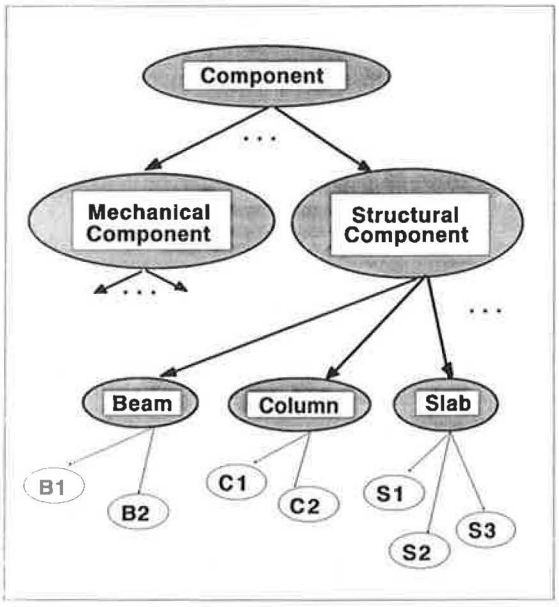

FIGURE 2 Classes and instances.

ODEPSI

Design Philosophy: Data Base plus Programming

The formula data base plus programming sums the objective of much current OODBMS research (*13–17*). This formula is also the design objective of ODEPSI. The OODBMS design philosophy resolves the impedance mismatch problem through the tight interaction of data base concepts within the OOP environment (*18*). Impedance mismatch is a term describing the language difference between a DBMS and related applications programs. Interfaces, like SQL, are widely used to resolve this language difference.

Traditional language and data base interfaces depend on a set of function calls or a separate data manipulation language that has little or no interaction with other language features. Traditional approaches burden the application program during data base interfacing and suffer more when they try to interpret data semantics. If the data base system is developed within the same programming environment as the application development, a programming environment that addresses these interfacing problems can be provided.

System Requirements

The primary requirement for ODEPSI is to provide a formal data (object) definition language (DDL) and data manipulation language (DML). ODEPSI provides extensive data manipulation capabilities, including most DBMS-like data access and query. This permits a focus on those operations that are application-specific. For a construction application, the user of ODEPSI needs to concentrate on the algorithmic behaviors of data objects like CPM processing or Earned Value generation. Access and query behaviors of each object are provided by the system.

ODEPSI also emphasizes automatic constraint management and semantic modeling. The system provides facilities for defining constraints on property values, object types, and object relationships. These constraints are then abided by the system and enforced to the data base users. Semantics of the real-world subject are modeled through the use of object properties, constraints, and interobject relationships. These capabilities are essential features for realizing the integrated data base of design and construction.

Objects live beyond the user sessions in which they are created. Each object is given a unique address by the system. As a concept-proving tool of the proposed research, ODEPSI needs only to be a single-user, virtual-memory–based, single-processor system. Other important data base issues, such as security control, concurrency control, and query optimization, are beyond the scope of this work.

System Development

Overall Architecture

ODEPSI is a set of new methods and classes added to the existing Smalltalk-80™ system. ODEPSI uses existing data

types, variables, and control structures of Smalltalk-80™. The specific additions made to Smalltalk-80™ were DDL and DML modules and a utility module for general system support, including automatic constraint enforcement and semantic modeling.

As schematically shown in Figure 3, ODEPSI and its generated data objects reside within the Smalltalk-80™ image. By combining all elements of program, DBMS, and data base within one homogeneous environment, the level of automation and integration necessary for integrated design and construction data management can be achieved.

ODEPSI Classes

ODEPSI is built around existing Smalltalk-80™ classes. As shown in Figure 4, most ODEPSI methods are created as a part of the Object and Behavior classes of Smalltalk-80™. Because these two classes are well structured and readily provide a rich set of generic methods for object programming, it is appropriate to develop and include ODEPSI's methods for object management within the same classes.

To handle methods that have no clear links to the Smalltalk-80™ classes, two new classes, Property and Db, were created. They were designed as subclasses of Object to inherit its generic behaviors. Property creates instances for class properties, and Db organizes all the classes and their instances created by ODEPSI. The Db class is the root node of all classes created by ODEPSI. Db also provides methods for maintaining and providing unique symbolic identifiers for the objects created.

Concepts

Defining Object Values Through Properties An object is a multivalued entity. Its values are explicitly defined and stored in terms of properties. Each property identifies and stores one or more values of the object. Each property of an object has six generic slots. The actual storage for these slots is produced by generating an instance of Property class.

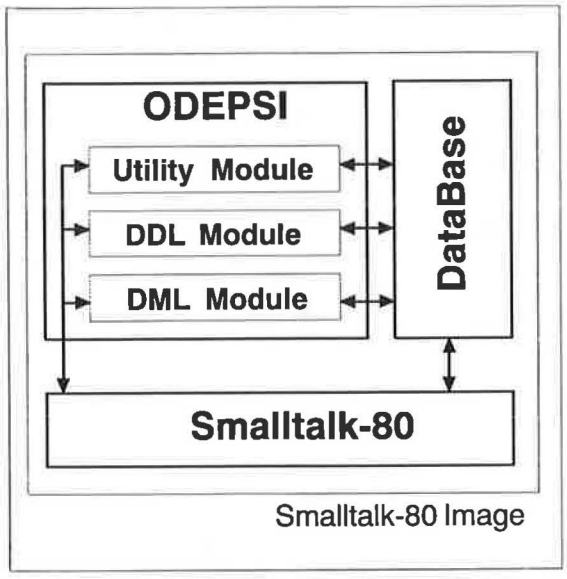

FIGURE 3 ODEPSI architecture.

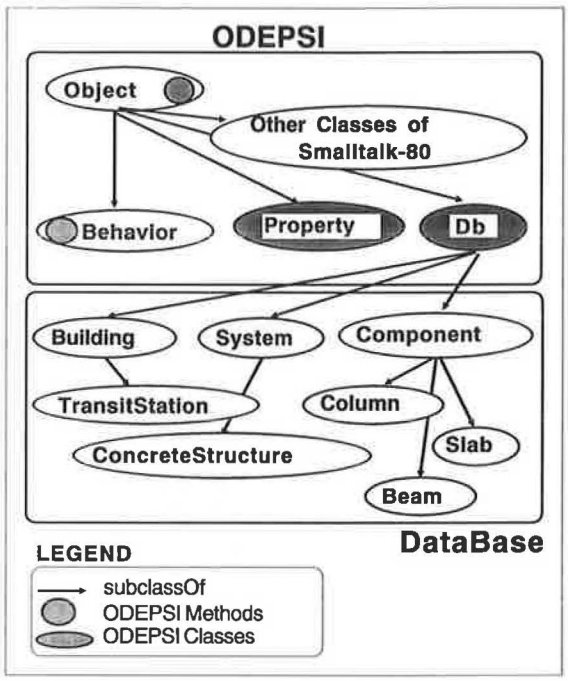

FIGURE 4 ODEPSI class structure.

Slots of a property are used to include the property's identification, value, allowable domain (type), constraint, trigger, and an indication of a single or multiple values. These slots are represented with the following Property instance variables: propName, propValue, propDomain, propConst, propTrig, and isSet (see Figure 5).

A class property in ODEPSI is internally represented with two items: an instance variable named after it and an instance of the Property class (see Figure 5). An instance variable is created to provide each instance a memory space to store its property value. A Property instance is created and allocated to the class to store the six generic contents of the property. A class object stores its Property instances into a class variable whose symbol is made by combining the class name and a word "PropertySet." As shown in Figure 5, Column's class variable is named "ColumnPropertySet."

Controlling Properties With Constraints A constraint can be defined for each property. A constraint of a property is used to check input data validity. A property's constraint is initially defined by the user with a list of procedural code. The constraint is compiled and stored in the constraint slot of the property. This list is evaluated when the property receives input data.

Figure 6 shows an example of this property constraint. A constraint is defined for the location property of Column class. A column instance, *C1*, is also created. The constraint of location property is evaluated when *C1* receives a value for its location property. As shown in the example, when a proper value, (1 2 1), is entered for *C1*'s location, the system accepts

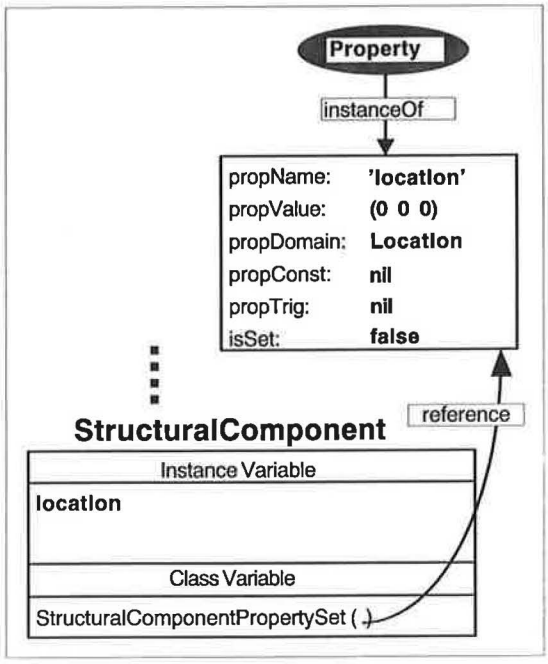

FIGURE 5 Defining a property location.

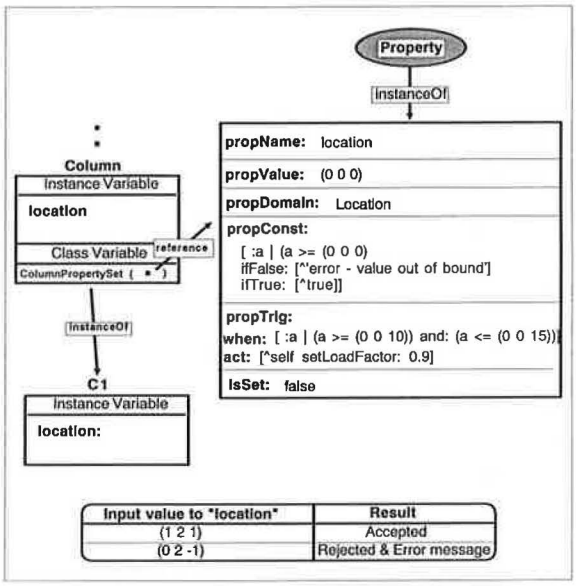

FIGURE 6 Property constraint and trigger.

the value—meaning that the constraint is satisfied. However, when an out-of-bound value, e.g., (0 2 −1), is entered, an error message is issued and the system rejects the value.

This property constraint maintenance mechanism is an important tool for enhancing the integrity and consistency of a data base. As stated earlier in the design philosophy, in this way the user of the data base can concentrate on more important aspects of problem solving. The data base supports the user in maintaining data integrity and consistency.

Controlling Properties With Triggers Another tool that can enhance the integrity and consistency of a data base is the trigger mechanism. A trigger can be defined for each property. A trigger is defined by the user like the constraint mechanism and stored in the trigger slot of a property. The purpose of the trigger mechanism is slightly different from that of the constraint. A property's trigger is activated only when its predefined condition is satisfied by the input data.

Figure 6 also provides an example of trigger operation. Here, a trigger is defined for the location property of Column class. The code defined in the "when:" part is the trigger's condition. If a column is between the 10th and the 15th floor, then the load factor of the Column is reset as 0.9. Neither of the two location values tried earlier satisfies this condition. Thus, this trigger is not activated.

Property Value Inheritance A class's properties are inherited to its subclasses. As shown in Figure 7, a StructuralComponent's subclass, Column, inherits all the properties defined for the StructuralComponent. When it is necessary for the Column class to change or customize the inherited properties, the user redefines any of the property content; value, domain, constraint, or trigger. This operation is supported by the system in the following manner: when the system faces this situation, it copies the property and assigns it to the class variable of the subclass, where the modification attempt is made.

For instance, Column's inherited property, load, is modified by overwriting its contents onto the copy of the property. The original load property object of the StructuralComponent is intact. Once the change is made, the Column class and its instances reference their own copy of the load property, not the property of the StructuralComponent class. This ability to customize the inherited properties is essential to efficiently delegate and control object behavior.

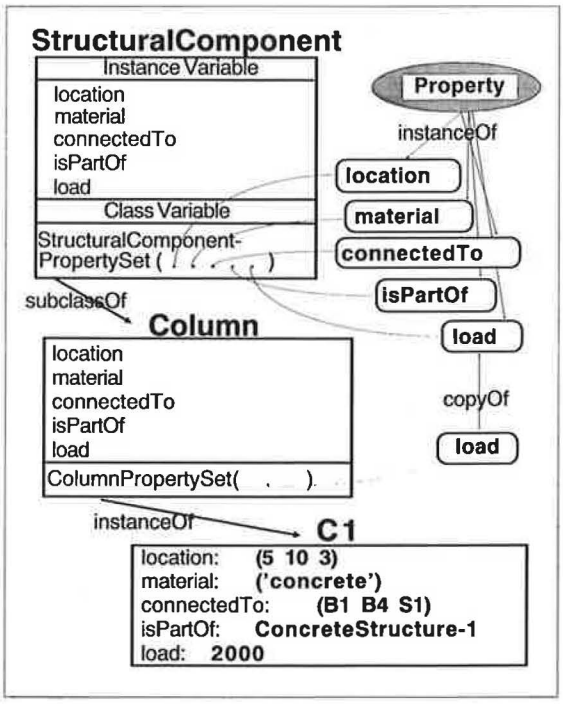

FIGURE 7 Property inheritance.

The benefit of this approach is shown in Figure 7. The Column class modified its inherited load property to define a constraint and a trigger. The constraint and trigger conditions required by Column may be different from those of the Beam or Slab classes. This approach allows individualized class behavior control. A Column instance, C1's load value is then affected by the constraint and trigger newly defined for its class.

SYSTEM USE

Creation of a Class Object

To create a new class object, a class method called defSubclass: is sent to an existing class from which the new classes are to be created. The following sequence of statement creates the initial definition of a new class, StructuralComponent, and its subclass, Column:

1. Component defSubclass: 'StructuralComponent'.
2. Structural Component defProperty: 'location';
   - defProperty: 'material';
   - defProperty: 'connectedTo';
   - defProperty: 'isPartOf';
   - defProperty: 'load'.
3. StructuralComponent defSubclass: 'Column'.

Component is the superclass of StructuralComponent. Once a new class named "StructuralComponent" is created, properties of the class are defined with the method defProperty: Another class, Column, is created as a subclass of the StructuralComponent class. As shown in Figure 7, the Column class inherits the properties of its superclass StructuralComponent. Each class in ODEPSI describes its status in terms of properties. This concept of property is similar to that of the attribute of a frame-based Artificial Intelligence (AI) knowledge representation concept.

Defining Set Properties

A property may have multiple values. For instance, a load-bearing wall may be built with several materials, such as concrete and rebar. To allow a property to have multiple values, it must be declared a set property using the class method defPropertyAsSet: The following statement defines the material and connectedTo properties of Column as set properties:

```
StructuralComponent defPropertyAsSet: 'material';
  defPropertyAsSet: 'connectedTo'.
```

The set property concept is an essential requirement for capturing interobject relationships, especially those of aggregation and association. For example, the Column instance is associated with several objects, such as Beam and Slab, and it is built with several materials (e.g., concrete and rebar). The Column is a subclass of the StructuralComponent class and, therefore, any decisions made on the StructuralComponent's properties are also inherited by the Column.

Defining Property Domain

A property value can be restricted to a certain class domain in ODEPSI. This ability allows tighter control of data semantics. The domain concept is similar to the type constraint of conventional data bases, but domain is used optionally when necessary. Unless defined, the system provides a default domain value of nil. The following statement defines the domain of the property isPartOf to the class ConcreteStructure:

```
StructuralComponent defPropertyDomain: 'isPartOf' with:
  ConcreteStructure.
```

This statement restricts the value of the isPartOf property of StructuralComponent to be a ConcreteStructure instance or that of its subclasses. Once a property is given a domain, the system automatically checks for the validity of its value whenever the property receives a value. Any Smalltalk-80™ class or user-defined class can be a domain value.

Defining Default Property Value

As an example, consider adding a new property to the Column class. This operation can be done by evaluating the following statement:

```
Column defProperty: 'width'.
```

A default value of the width property can also be defined. The following statement sets the property width of the class Column to 12:

```
Column defPropertyValue: 'width' with: 12.
```

Default property values of a class are automatically propagated to the class instances. For example, all instances of Column will have their width properties valued as 12. If some instance object's width is not 12, then the user must input the new value explicitly. This default concept is useful for engineering applications in which many standard values are repeatedly applied. Default values of class properties are also inherited to subclasses.

Defining and Triggering a Property Constraint

Knowledge necessary to propagate design changes to construction data can be captured through the use of these mechanisms. For example, if the user wants to limit allowable load values of the Column between 2,000 and 2,500 kg/in.^{2}, the user would define this constraint using the class method defPropertyConst::

```
Column defPropertyConst: 'load'
  with: '[ :a | (a =< 2500 and: a => 2000)
  ifFalse: [^"error:
  load value should be between 2000 and 2500"]]'
```

In ODEPSI, both constraints and triggers are represented and stored within the Block object shown. An instance of Block

is surrounded by square brackets and evaluated when it receives the message value. A detailed explanation of the operational aspects of Block and ifFalse: is available from Goldberg and Robson (*19*).

Continuing with the example, suppose the load value is increased from 2,000 to 2,500 kg/in.^{2}. A likely scenario is that the material estimate, especially rebar and perhaps formwork, may have to be adjusted, because the increased load-bearing capacity may demand less structural reinforcement. Users can prepare for this situation using triggers. The following trigger is defined to update the effect of the load-bearing capacity increase:

```
Column defPropertyTrigger: 'load'
  when: '[ :a | (a > 2000)
  act: [^self updateMaterialEstimate ''rebar'' with: a]]'
```

The preceding message defines the property load trigger and stores it in the class Column. Whenever the property load value of any one class instance is updated and meets the condition (defined in the when: message), its trigger (defined in the act: message) is fired.

SEMANTIC MODELING

Four basic relationships—generalization, specialization, aggregation, and association—are represented and supported in ODEPSI. The ability to handle these basic relationships is what distinguishes OODBMS from other data base concepts. To avoid redundancy, specialization and association are not explicitly discussed here. Their meaning can be inferred by understanding the generalization and aggregation concepts.

Generalization

A generalization relationship permits the grouping of similar objects into a single unit; in other words, generalization describes objects in an abstracted form. For example, the class Column may be a generalization of more detailed classes such as ConcreteColumn, SteelColumn, or WoodColumn. The class Column may store generic properties common to all columns. All columns have properties, such as columnDimension, columnLocation, or supportingComponents. The class Column is an ideal candidate for accommodating all these properties. Then more detailed properties can be assigned to the specialized classes. For example, the class WoodColumn may include a property like woodType to describe the wood material that it is made of. The following statements describe these generalization relationships with ODEPSI syntax:

```
StructuralComponent defSubclass: 'Column'.
Column defProperty: 'columnDimension';
  defProperty: 'columnLocation';
  defProperty: 'supportingComponents'.
Column defSubclass: 'ConcreteColumn'.
Column defSubclass: 'SteelColumn'.
Column defSubclass: 'WoodColumn'.
WoodColumn defProperty: 'woodType'.
```

One important aspect of the generalization relationship is its implication for inheritance. In ODEPSI, properties and their values of superclasses are inherited to their subclasses. In the previous example, those classes specialized from Column inherit all the properties of Column. In addition, those classes inherit all properties of the superclasses of Column (e.g., StructuralComponent). Like the WoodColumn class, they can add properties to themselves.

Aggregation

An aggregation is a form of an isPartOf relationship. With aggregation, one class that contains or aggregates others assumes the role of assembly, and the other classes assume the role of components. Aggregation relationships are especially abundant in engineering applications. A building's structural frame is an aggregation of basic structural components, such as columns, beams, slabs, or stairs.

An aggregation relationship between Floor-1 and the floor components (Room-1, Room-2, Room-3, and Corridor-1) is established as follows:

```
Floor-1 setPropertyValue: 'contains' with: Room-1;
  setPropertyValue: 'contains' with: Room-2;
  setPropertyValue: 'contains' with: Room-1;
  setPropertyValue: 'contains' with: Corridor-1.
```

CONCLUSIONS AND RECOMMENDATIONS

An experimental OODBMS, ODEPSI, has been presented. Basic concepts of object-oriented data representation and management have been discussed. Issues involved in ODEPSI's implementation and its basic syntax have been presented. The approach to achieving property value inheritance and semantic modeling has also been discussed.

The initial results of this work indicate that a homogeneous environment for programming and data base is advantageous for representing the semantics of a construction project. A project model that integrates design objects with construction planning and control objects is being developed with the current version of ODEPSI. This exercise will further clarify directions to take in developing an object-oriented project planning and control system.

The ODEPSI experience indicates several improvements to its functions. At the top of the list is a composite object function. To better represent functional and spatial constraints of a facility, an explicit mechanism is needed to handle those objects that are highly aggregated (i.e., assembly-type objects).

Another useful function being worked on is object versioning combined with the time concept. In the reality of design and construction, many objects are changing property values while design and construction are in progress. On many occasions, histories of those objects must be maintained to resolve conflicts or to capture valuable engineering or construction information that would otherwise be lost.

In the future, composite object management capability will be added to ODEPSI. As more of these enhancements are

developed, transportation managers and engineers will see the appearance of object-oriented data bases in day-to-day practice.

ACKNOWLEDGMENT

This material is based on work supported by the National Science Foundation (NSF).

REFERENCES

1. *Report from The 1986 Workshop on Integrated Data Base Development for The Building Industry*. Building Research Board, Commission on Engineering and Technical Systems, National Research Council, Woods Hole, Mass., June 1987.
2. Database to Track Project Progress. *Engineering News-Record*, Feb. 1988, p. 26.
3. K. C. Choi and C. W. Ibbs. *Cost Effectiveness of Computerization in Design and Construction*. Technical Report 17, Department of Civil Engineering, University of California at Berkeley, 1989.
4. E. J. Yannakoudakis. *The Architectural Logic of Database System*. Springer-Verlag, London, 1988.
5. O. J. Dahl and K. Nygaard. SIMULA: an algol-based simulation language. *Communications of the ACM*, Vol. 9, 1966, pp. 671–678.
6. T. Kaehler and D. Patterson. A Small Taste of Smalltalk. *BYTE*, Aug. 1986, pp. 145–159.
7. G. A. Pascoe. Elements of Object-Oriented Programming. *BYTE*, Aug. 1986, pp. 139–144.
8. B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.
9. Smalltalk/V286 Tutorial and Programming Handbook. *Smalltalk/V286*, Digitalk Inc., 1988.
10. P. P. S. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, March 1976, pp. 9–36.
11. J. M. Smith and D. C. P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, Vol. 2, No. 2, June 1977, pp. 105–133.
12. L. A. Rowe and M. R. Stonebraker. The POSTGRES Data Model. *Proc., 13th International Conference on Very Large Data Bases*, Brighton, England, 1987, pp. 83–96.
13. A. Purdy, B. Schuchardt, and D. Maier. Integrating an Object Server with Other Worlds. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987, pp. 27–47.
14. J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987, pp. 3–26.
15. A. Ege and C. A. Ellis. Design and Implementation of Gordion, an Object Based Management System. *Proc., IEEE 3rd International Conference on Data Engineering*, Los Angeles, Calif., Feb. 1987, pp. 226–234.
16. J. Diederich and J. Milton. ODDESSY: An Object-Oriented Database Design System. *Proc., IEEE 3rd International Conference on Data Engineering*, Los Angeles, Calif., Feb. 1987, pp. 235–244.
17. M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for EXODUS. *Proc., ACM SIGMOD International Conference on Management of Data*, 1988, pp. 413–423.
18. G. Copeland and D. Maier. Making Smalltalk a Database System. *Proc., ACM SIGMOD International Conference on Management of Data*, 1988, pp. 316–325.
19. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass., 1983.

Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not reflect the views of the National Science Foundation.

Publication of this paper sponsored by Committee on Construction Management.