A Data Model for Design Objects

Muruganandan Kumar
and
Johnny Wong
Computer Science Department
Iowa State University
Ames, Iowa- 50011

Abstract

Modeling concepts for handling structural data of hierarchically composed complex objects are presented. A complex object is modeled as an hierarchy of interfaces and an implementation. Objects sharing the same interface at any level have same attributes and attribute values up to that level in the hierarchy. This structuring facilitates storage of multiple object versions and their subsequent reuse in design databases, where multiple versions of design artifacts are routinely created and maintained. Other important concepts needed for design databases are presented, and in particular two types of instantiation - parametric instantiation, and conditional instantiation - are proposed.

1 INTRODUCTION

The Relational model (R-model) has been found inadequate for advanced database applications that are now expected of databases in areas such as CAD/CAM, office information systems, and software engineering. The inadequacy is primarily due to the following three reasons. First, conventional relational database systems with flat tuples do not provide adequate structuring mechanism to capture all information pertaining to an object. Second, even relatively simple applications in the non-traditional fields pointed out earlier, require computation of transitive closure. However, it is well known that transitive closure cannot be expressed by a relationally complete query language [AU79]. Third, simple tables of atomic components are not capable of capturing a large class of data encountered in such advanced database applications (e.g., text).

Attempts at solving the problems associated with the R-model can be classified into three groups:

1. Extensions to the R-model [Cod79],[HL82].
2. Non conventional data-models including the Entity Relationship model (E-R model) [Che76] and works related to non-first normal forms(NF^2) [AB84], and more general models such as [BK86].
3. Object oriented approaches; efforts in this area are based on object oriented programming language paradigm. The distinguishing features of this paradigm are attribute inheritance, class hierarchy, and message passing [Dit87],[KL88].

The first normal form (1NF) structuring constraint imposed by R-model forces a database designer to decompose complex objects into a set of homogeneous tuples, obeying the various dependency constraints. In the process the model loses the ability to express the schema in an easily understandable way. In NF^2 models, the constraint that tuple attributes be atomic is relaxed allowing attributes to be relations. The R-model retains the structuring concept: the relations, with its operations extended to manipulate the complex structure. However, this approach also fails to capture the schema of the world of discourse in an easily understandable way.

In recent years, researchers have focused on object-oriented paradigm to model complex objects. Although several prototypes have been developed, it is unclear whether a user friendly interface can become available in the near future. Meanwhile the E-R model can be viewed as the first step in the direction of object-oriented models. E-R model offers the concept of *entities* (E) and entity sets to represent the objects of interest, along with *relationship* (R) and relation-

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi
©World Scientific Publishing Co.

ship sets, distinct from entities, to capture the association relating the entities. The third distinct concept is that of *attributes* which are distinct from entities and relationships, and are used to express the characteristics of the entities. The E-R modeling approach offers the designer a natural way for representing the 'world of interest'. However, current E-R models share the same inadequacy of R-model to represent complex objects.

A modeling technique based on E-R model for complex objects called *molecular aggregation* has been recently proposed [BB84]. Other modeling concepts needed for nested (composite) objects such as object interfaces, object versions, and instantiation have been presented in a recent work [BK85]. Here, we extend the modeling techniques and propose a scheme suitable for design objects. In particular, we use VLSI and electronic circuit design as examples throughout to illustrate the concepts presented. In addition, we introduce the notion of *parametric instantiation* and *conditional instantiation* which will in most cases alleviate the problem of change propagation and change notification in design databases. While the primary focus of the proposed scheme is to encapsulate the structural information of a complex object, rather than its behavior, the scheme can be adopted for modeling geometric and structural data of other engineering artifacts as well.

The remainder of this paper is organized as follows. In the next section we briefly examine the requirements of a design database model. In section 3 we present the concepts needed for representing complex objects and their versions. In section 4 and section 5 we present other concepts needed for modeling complex objects. Section 6 deals with a set of primitive operations for manipulating complex objects and section 7 concludes the paper.

2 REQUIREMENTS OF A DESIGN DATA MODEL

In this section we present some of the issues that need to be resolved for an efficient implementation of a design database.

1. Complex Objects: The design artifacts usually are composed of a collection of heterogeneous records which together represent the design artifact - a complex object. The set of records cannot be conveniently represented by a small set of homogeneous records or tables of tuples as in the case of conventional database entities.

2. Composite Objects: Composite objects are complex design objects formed by integrating simpler objects that are defined independently, resulting in a composition hierarchy. The data model should facilitate the creation and maintenance of such a composition hierarchy. Note that all composite objects are complex objects as well.

3. Sharability of Objects: For practical considerations such as economy, efficiency of design effort and storage, composite objects are designed using components that are defined and available in a design database accessible for designers. The components in the library are shared by all designers. The model should facilitate such sharing. In addition, a composite object might use several instances of an object in its design. Therefore, during representation, such redundancy needs to be avoided.

4. Multiple Versions: Typical design processes involve creation and testing of multiple versions - both as design alternatives and as revisions to existing design. Thus several versions may exist for each 'object type'. The data model should facilitate the creation and maintenance of such a composition hierarchy.

5. Abstraction Hierarchy: In a system where there are users at different levels with different views of the data, a hierarchy of data abstraction is essential for the users to access and manipulate the objects. In addition to the ease of use in handling and managing the complex data, an abstraction hierarchy allows for the stability of the system. Each abstraction level acts as a firewall for changes in the data as they evolve - which is often the case in a design database where all changes at lower level may not propagate up in the data hierarchy.

6. Operations on Objects: The operations on the design objects also pose some problems. The complex objects may have to be retrieved and manipulated as a whole or a part of the object may be modified. In both cases, the data model should facilitate the operations efficiently.

7. Multiple Representation: An object may have different interpretations in different 'views'; for example a circuit diagram could be viewed as a set of connectivity information or a graphic layout, depending on the environment.

8. Schema Changes: Unlike conventional databases, the schema of the design database changes frequently as

the structure of the design objects change. Consequently, design databases should facilitate and support schema evolution.

In this paper, we address all the above issues except those concerning multiple representation and schema evolution. The problem of multiple representation is application dependent and needs to be handled during interface design.

3 COMPLEX OBJECTS

Design artifacts are complex objects and they can be represented by a set of heterogeneous tuples. The object can be abstracted and represented by a single tuple (with its unique object id among other attributes) at a higher level. The various attributes of the object can be represented by a set of inter-related tuples or tables of tuples at lower levels. Several proposals have been made to extend conventional database systems to include the notion of complex objects [LP83], [KLMP84], [BK85].

In a recent work [BB84], complex objects called molecular object are formed by the abstraction technique referred to as *molecular aggregation*. Molecular aggregation is the abstraction of a set of entities and their relationships into a higher level entity.

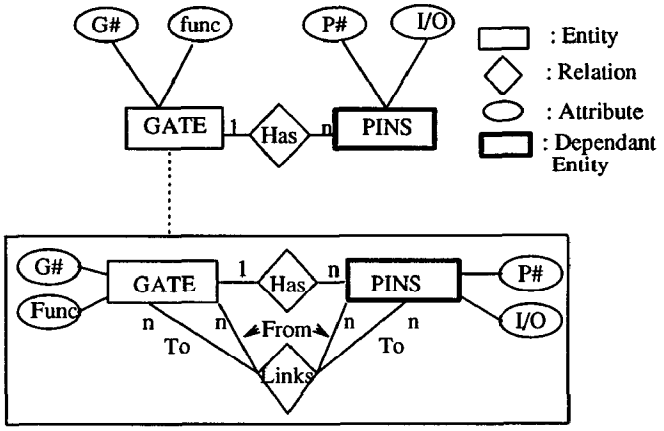


Figure 1: Molecular aggregation

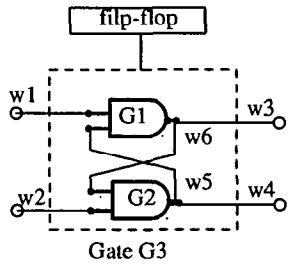


Figure 2: Molecular aggregation of gate: Flip-Flop

Using E-R modeling technique [Che76], a complex object can be reduced to a set of tables of relations. At the top level an object is denoted by a single tuple identified by its unique identifier (oid) in an object-table, along with a set of descriptive attributes $OA_1,...,OA_n$. At the lower level, tables $T_1,...,T_m$ hold other tuples that describe the object in finer detail. Tuples in table T_i have key oid.K_i. See Figure 1 for an abstraction of a composite object gate which is made up of other objects (gates) and relations (wires). If the component entities in a complex object are themselves 'complex', then each of the components can again be further defined in terms of other simpler objects until all attributes are expressed in terms of a set of primitive objects. Figure 2 shows a gate 'flip-flop' which is made of two nand gates and a set of relations representing metal-runs or connections.

Table 1: Gate

Gate#	Desc	No.pins	Parent
G1	2-nand	3	G3
G2	2-nand	3	G3
G3	flip-flop	4	-

Table 2: Pins

Gate#	Pin#	I/O
G1	p1	I
G1	p2	I
G1	p3	O
G2	p1	I
G2	p2	I
G2	p3	O
G3	p1	I
G3	p2	I
G3	p3	O
G3	p4	O

Table 3: Links

Wire#	From		To		Parent
w1	G3	p1	G1	p1	G3
w2	G3	p2	G2	p2	G3
w3	G1	p3	G3	p3	G3
w4	G2	p3	G3	p4	G3
w5	G1	p3	G2	p3	G3
w6	G2	p1	G1	p3	G3

The attributes of the flip-flop and its components can now be expressed by a set of tables: Tables(1-3).
In the case of Figure 2, we assume that 2-nand gates and wires are primitive objects that need not be redefined further.

4 INTERFACE AND IMPLEMENTATION

The scheme presented earlier was capable of capturing the complex relationship between different attributes of a complex object. However, such a scheme when used in a design environment will have several shortcomings. For example, if another version of an object implementation is designed (a flip-flop with one of its wires re-routed), then it would entail storing all the information associated with the new version as well, much of which is a repetition of the data associated with the previous version.

The second problem is in accessing the complex objects. The objects may be used in the design of other complex objects. In such cases, the information that is pertinent to the user are only a subset of the attributes of the object used. In general, the abstraction of the implementation is sufficient for a user to use a complex object. For example, in the case of integrated circuits, the pin configuration and electrical characteristics are sufficient for a designer to use a particular chip in a design. Therefore, it may be unnecessary for a designer to view the complex object in its entirety which contains details outside the scope of his interest. The projection of the implementation of the complex object can be captured by the *interface* of the object. The interface is the abstraction of the implementation, and the object interacts with the external world (other objects) through the interface.

Object interfaces can be viewed as the abstraction of the object implementation. Thus each object definition has two parts: the object interface and the object implementation. We now apply the concept of generalization abstraction to object interfaces. In design databases, as we pointed out earlier, several versions of an object type may be designed (as design alternates or refinements). The different versions may have the same interface features and they differ only in the implementation specification. Thus the interfaces of different versions of the object may now be merged and represented by a single interface – an object which holds data common to the different versions of the object. In Figure 3, for example, two versions of flip-flop, each with its own interface and implementation are shown. Except for the routing of one of the wires, most of the features in both the objects are identical. Thus the common part of the attributes they share is extracted out and placed in the object 'interface' while the individual versions retain the attributes that are not shared. The association between the implementation and the interface can be maintained by the relation "version-of" which relates interface-types with its versions. In our work, this relation is denoted implicitly by including the column "Interface/Type" with the table denoting the entity "version". It has to be noted that an interface of an object can itself be a complex object as it may have complex attributes. For example, the electrical characteristics of a gate can be represented by a set-valued attribute :"electrical-char." This complex attribute can be represented by a table with a scheme (Char#, Char).

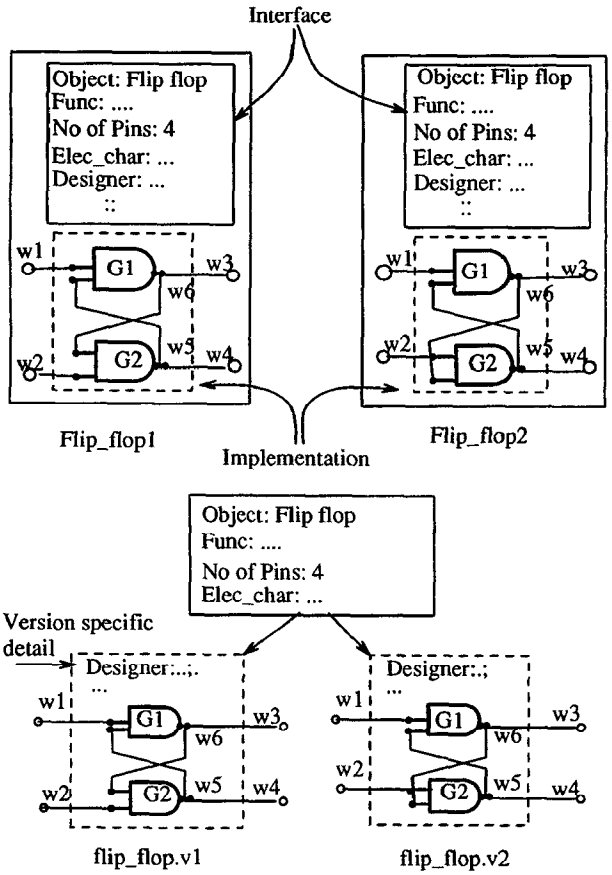


Figure 3: Object implementation and interfaces

4.1 Hierarchy of Interfaces

An object can be represented as the sum of its two parts: its interface, and its implementation. The objects that share the same interface but different implementation can be grouped together and the interface information is generalized resulting in a class of interfaces. All implementations of an interface inherit all information of the interface. If one of the attributes of the interface changes, then it results in a new interface (class). However, in a design database designers often create several versions of design objects as design alternates and as design revisions, possibly with different interfaces as well resulting in a profusion of interface types.

As pointed out in section 2, design data have to be represented in several levels of abstraction. In addition to the stability of the database, such structuring facilitates systematic database design, efficient implementation, and economy of storage.

Two well known forms of data abstraction are aggregation and generalization [SS77]. Aggregation abstraction referred to a form of abstraction where a relationship between a set of objects is viewed as a higher level object in which some of the details may be suppressed. For example, the relationship between the entities Husband:John, Wife:Mary, License-no:1211 and Date:Junel,1990 may be abstracted to 'marriage'. In generalization abstraction a set of similar objects are grouped into a generic object where the members of the set are instances of the generic object. For example, a set of married couples can be abstracted to a generic object 'couples', where differences between individual members of the set are overlooked. The R-model exhibit both forms of abstractions. However the conventional R-model is insufficient to model complex objects.

The interface concept is extended to a hierarchy of abstractions, where at each level objects that share the same attribute values are grouped and represented as a single object at that level. Each class object (interface) inherits the attributes of the class in the hierarchy above, of which it is a member. That is, class objects at each level is a generalization of the member objects.

4.2 Attribute Types

Entities are identified by the set of attributes associated with them. As in the case of R-model, the E-R model allows atomic valued attributes in the relations. We distinguish the following attribute types associated with an object. (1) Inherited attributes: attributes inherited from its parent class. These are non-modifiable. (2) Machine assigned attributes: fixed and non-modifiable (e.g., version number, creation date). (3) Environment dependent attributes: Attributes that are inherited from the environment. (e.g. size of font in a text, location of a cell in VLSI circuit). (4) Local attributes: These are version dependent attributes that are local to the version. The environment dependent attributes or parametric attributes can be associated with the interface resulting in a parametric interface or with a version resulting in a parametric version. In both cases, the domain of the parameter need to be specified. The attributes can then be used in the implementation as in the case of formal parameters in programming languages. This is further discussed in subsequent section.

5 INSTANTIATION

Once an object is specified, then it can be used in the implementation of other objects by *instantiating* the object. Object instantiation allows for reusing or sharing of objects. An instance of an object inherits all the attributes of the object version along with the attributes specified in the interface. In addition multiple instances of the same object may be instantiated in the design of a single complex object. Instances of an object version may have their own instance specific attributes that are assigned during instantiation of the objects. The different instances are distinguished by associating an instance number along with the identifier sequence for the version number (interface name, version number). We assume that interface names are unique system-wide. Instances of object are related to the object implementation by a 'Inst' relationship which is in fact a 'is-a' relationship. The E-R diagram for a gate instance flip-flop is shown in Figure 4.

Tables 4–8 show the instance-relation for a flip-flop using two instances of a nand gate.

Table 4: Gate Interface

Type#	Desc	$a_1 \ldots a_n$
T1	2-nand	...
T2	flip-flop	...
T3	2-nor	...
..	..	..

Table 5: Gate-version

Type#	Version#	$b_1 \ldots b_m$
T1	V1	...
T2	V1	...
T3	V2	...

Table 6: Table Gate-instance

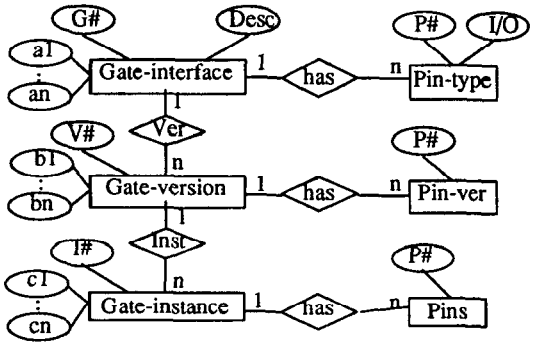
Type#	Version#	Instance#	parent	
T1	V1	gi1	T2	V1
T1	V1	gi2	T2	V1
T1	V1	gi1	T2	V2
T1	V1	gi2	T2	V2

Table 7: Pin-type

Type#	Pin#	I/O
T1	p1	I
T1	p2	I
T1	p3	O
T2	p1	I
T2	p2	I
T2	p3	O
T2	p4	O
..	..	..

Table 8: Links

wire#	Parent		From				To			
	Type	Version	Type	Version	Inst	Pin	Type	Version	Inst	Pin
w1	T2	V1	T2	V1	-	p1	T1	v1	gi1	1
w2	T2	V1	T2	V1	-	p2	T1	v1	gi2	2
w3	T2	V1	T1	V1	gi1	p3	T2	V1	-	3
w4	T2	V1	T1	V1	gi2	p3	T2	V1	-	4
w5	T2	V1	T1	V1	gi1	p2	T1	V1	gi2	3
w6	T2	V1	T1	V1	gi2	p1	T1	V1	gi1	3



a1..an: interface attributes

b1..bn: version specific attributes

c1..cn: instance specific attributes

Figure 4: Object instantiation

5.1 Parametric Instantiation

Object interfaces and implementation can be specified using objects that are supplied as parameters during instantiation. If an interface has a parameter, then it is said to be a parametric interface. Every implementation of the interface inherits the parameter that is supplied to the interface during instantiation. The parameters of the object can also be expressed in tables.

An implementation can also be parametrized if the implementation uses parameters in the design. The parameters are supplied when a version is instantiated. Figure 5 shows the E-R model of a parametric implementation. The relation 'Actuals' and 'Formals' capture the association between the various instances of the objects involved.

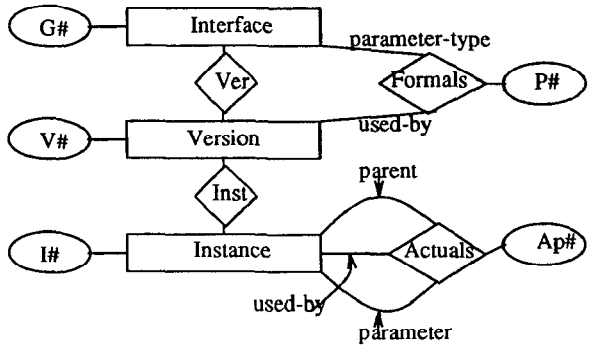


Figure 5: Parametric implementation

Figure 6 shows a specification of a flip-flop which uses two nand gates. The type of the parameters (interface) are specified while the actual version is left unspecified in the design. Users of the flip-flop implementation select a version

```
Object P_flip-flop is_a  T2
       version : V2
    local attributes: ..
       created-by : ..
             .... : ...
   components:
       FP1 : is_a T1;
       FP2 : is_a T1;
   relations :
       wire1: From: self.p1; To: FP1.p1
       wire2: From: self.p2; To: FP2.p2
       wire3:..
       ......
end {object P_flip-flop}

Object counter is_a T4
{T4 is an interface for a binary counter}
       version : V1
    local attributes: ..
       created-by : ..
    components:
       gi3: T2.V2(T1.V1.gi1,T1.V1.gi2)
      {counter is the parent}
      {T2.V2 is a  P_flip-flop; T1.V1 is a 2-nand}
        ...
        ...
end {object counter}
```

Figure 6: A Parametric flip-flop

Table 9: Formals

Fp#	used-by		parameter-type
	Type	Version	
fp1	T2	V2	T1
fp2	T2	V2	T1

Table 10: Actuals

Ap#	Parent			Used-by			Parameter		
	Type#	Ver#	Inst#	Type#	Ver#	Inst#	Type#	Ver#	Inst#
ap1	T4	V1	gi3	T2	V2	gi1	T1	V1	gi2
ap2	T4	V1	gi3	T2	V2	gi1	T1	V1	gi3

of nand gate and create two instances of the nand gate and pass them as parameters to the instance of the flip-flop. This allows the users to create versions of objects that have required characteristics by using a small set of 'template' versions and a set of base objects that are passed on as parameters. To capture the information in Figure 5 the set of tables is augmented by Tables 9–10.

5.2 Conditional Instantiation and Lazy Evaluation

In design environments, several design processes or activity may be in progress concurrently, co-operating to produce a final object. The final object is a composite of other complex objects, which may themselves be composite objects, thus resulting in a 'is-a-part-of' hierarchy. Designers of higher level objects (objects at the top of the composition hierarchy tree) have to delay their design until the lower level objects they intend to use in their design are specified. For example, if a designer wishes to use an object of type G in his design, then he has to wait for the correct version of the object G (the one with attributes that matches his specification) to become available so that a copy of the circuit can be instantiated in his design. This disallows concurrent design activity at all levels of the composition hierarchy.

Another problem with instances using explicit version name is the problem of inconsistency which arises when versions are modified or deleted. Due to the constant flux in the design databases, versions are periodically redesigned and new versions developed. Thus when composite objects are designed using instances of specific versions of other objects, if the object versions at lower level are modified then the composite object may not be consistent or correct. To maintain consistency of the composite objects, when an object O is modified, the changes have to be notified to other objects that use the object O in their design.

Let A be a composite object which has an instance of object B (refer to Figure 7). If the implementation of B is changed by its designer, then the design of object A may not be consistent if the design was based on some of the attribute values local to B. Two options have been suggested in literature for notification of change in design environments: flag based notification, and message based notification [CK86].

In both schemes, object A has to respond to the changes in B by verifying the design of A and revising the design if necessary. If the design constraints do not allow change in the

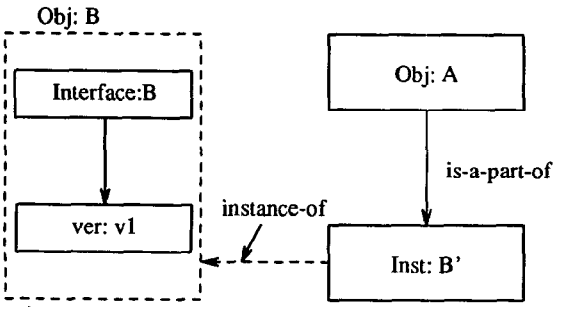


Figure 7: Change notification

attributes then object A becomes invalid. If the changes in B are within the design constraints of A, then the changes in B are reflected in the new design of A which has to be revised. This revision may result in changes in A's attributes, necessitating A to notify the changes to other objects that might use instances of A in their design.

The problems associated with modification of objects and change notification can be minimized or eliminated by using *conditional instantiation.* Under this scheme, objects can be instantiated without specifying the version explicitly. The instantiation is made by imposing constraints on the attribute values of the versions. Any version that meets the constraint can now be instantiated explicitly. Thus composite objects may be designed using objects that are still in the process of being implemented. The E-R diagram for conditional instantiation is shown in Figure 8.
The process of assigning an explicit version name to a conditional instantiation is called 'pinning'. When an object

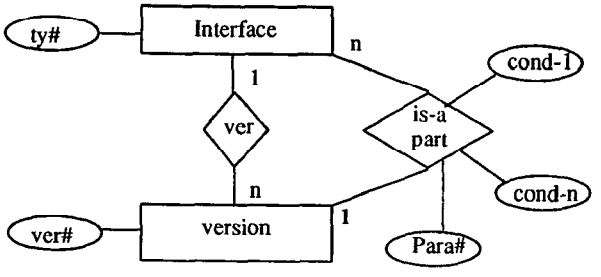


Figure 8: Conditional instantiation

is pinned, the constraints are evaluated to identify the exact version as and when required by the system. This can be done by performing a select on the attributes involved in the condition over the domain of the object-types involved. By keeping the versions un-pinned, the implementations can remain current and unnecessary change notification/propagation messages and action can be avoided.

6 OPERATIONS ON COMPLEX OBJECTS

A set of primitive operations are presented below using which other complex operations can be built. The operations are classified into three groups: (1) operations on object interfaces,(2) operations on object versions, and (3) operations on object instances. The operations assume the availability of templates or storage area in the user space which holds the object that is manipulated. In each of the operations, the system accesses the various relations (tables) involved to retrieve/store the tuples that constitute the complex object. The object structure (schema) is maintained in a database which is accessed first to obtain the set of tables that hold the complex object.

6.1 Operations on Object Interfaces

The following operations are informally defined on object interfaces. The parameter 'T-set' refers to the set of interfaces (super type) of which the interface under discussion (type) is a member. The user space that holds the interface is called T-temp, and the T# refers to the unique identifier for the interface.

1. new-interface(T-set,T-temp): Install the new interface defined in T-temp in the set specified and returns the T# assigned to the interface.

2. read-interface(T#,T-temp): Read the specified object interface into T-temp.

3. delete-interface(T#): Delete the object interface from the corresponding I-set. Note that the deletion of an object interface may create orphans out of the object versions that use the interface. Therefore, in the event of deletion of object interface, all object versions of the interface have to be deleted as well.

6.2 Operations on Object Versions

The operations allowed on object versions are listed below. The parameter V-temp refers to the area in the user space that holds the object version; V# is the machine assigned version number, and T# is the interface identifier.

1. create-version(T#,V-temp): Install the object version stored in V-temp as a version of interface T#; the call returns a version identifier V#.

2. read-version(T#,V#,V-temp): Read the specified version of the interface into the user area V-temp.

3. delete-version(T#,V#): Delete the named version from the set of versions of the object interface T#. As in the case of object interfaces, deletion of a version may require deletion of all instances of the version in the database.

4. update-version(T#,V#,V-temp): Replace the version V# with the new version in the V-temp. Composite objects that used the old version may now be inconsistent due to the changes in the version.

6.3 Operations on Instances

The following set of operations are defined for the object instances. The user area used for storing object instances is named I-area The instance identifier is I#.

1. install-instance(T#,V#,I-area): Store the instance of the version that is in I-area. The call returns the identifier I# associated with the instance of the object.

2. read-instance(T#,V#,I#, I-area): Read the specified instance into the specified area.

3. delete-instance(T#,V#,I#): Delete the specified instance of the object from the database.

4. pin(T#,V#,I-area): Evaluate the conditions of instantiation of the object specified and instantiate the necessary parameters.

7 CONCLUSION

Modeling concepts necessary for modeling complex objects - particularly, schemes necessary for storing data associated with design objects are presented. We use examples from VLSI and electronic circuits to illustrate the concepts. We propose the concepts of parametric versions that facilitates the reusability of a set of design objects - the parametric versions/ interfaces to obtain several versions or instances of the object type but with significantly different characteristics. We also propose the concept of conditional instantiation which can be used to maintain integrity even in the presence of changes in object versions.

In addition to the set of primitive operations suggested, a query language capable of supporting complex operations need to be developed to support the model.

References

[AB84] S. Abiteboul and N. Bidoit. Non first normal for relations to represent hierarchically organized data. *ACM PODS*, 191–200, September 1984.

[AU79] A. V. Aho and J. D. Ullamn. Optimal partial match retrival when fields are independently specified. *ACM Transactions on Database Systems*, 4(2):168–179, 1979.

[BB84] D. S. Batory and A. P. Buchmann. Molecular Objects, Abstract Data Types, and Data Models: A Framework. In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 172–184, Singapore, August 1984.

[BK85] D. S. Batory and W. Kim. Modeling concepts for VLSI CAD objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[BK86] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *ACM SIGMOD'86: International Conference on Management of Data*, pages 53–59, Washington, 1986.

[Che76] P. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CK86] H. Chou and W. Kim. A unifying framework for version control in a CAD environment. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Dit87] K. R. Dittrich. *Object-oriented database systems- A workshop report in Entity-Relationship Approach.* North-Holland, Amsterdam, 1987.

[HL82] R. Haskin and R. Lorie. On extending the functions of a relational database systems. In *Proceedings of ACM SIGMOD*, pages 207–212, ACM, Orlando, FL, June 1982.

[KL88] W. Kim and F. H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, NY, 1988.

[KLMP84] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, Singapore, August 1984.

[LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Proceedings of Database Week: Engineering Design Applications*, pages 115–121, IEEE, Los Angeles, CA, May 1983.

[SS77] J. Smith and D. Smith. Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.