MODELING CONCEPTS FOR VLSI CAD OBJECTS

D. S. Batory and Won Kim[1]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-35 December 1984

[1]Microelectronics and Computer Technology Corporation, Austin, Texas.

and object versions (interface and implementation) can be instantiated. Instantiation distinguishes an object (type or version) from its copies. Copies are simply reproductions of a master; the copies themselves are *not* versions.

Instantiation, like type-version generalization, has attribute inheritance. An instance of an object (type or version) inherits all attributes of that object. Instances also have attributes that are not inherited (e.g., coordinate positions, inputs and outputs that are specific to an instance). It is the non-inherited attributes which enable different instances to be distinguished. A precise definition of instantiation is presented in Section 3.3.

Parameterized versions is the fourth of our proposed modeling concepts. Again consider Figure 2. The implementation of the adder uses four adder-slice circuits (i.e., circuit interfaces). However, no implementation of any adder-slice circuit is specified. Either one or both of the adder-slice implementations of Figures 3b and 5 could be used in building an actual adder circuit.

Figure 2 can be understood as a template. Each instance of an adder-slice type (interface) defines a socket or hole which can be plugged by an instance of any adder-slice version (implementation). It is this notion of 'sockets' and 'plugs' ([EDI84]) which gives rise to the concept of parameterized versions. The parameterization of objects is shown to be a natural consequence of molecular objects, type-version generalization, and instantiation in Section 3.4.

3. Definition of Modeling Concepts

We will define and develop the concepts of molecular objects, type-version generalization, instantiation, and parameterized versions in the context of the ER model. We have chosen the ER model because of its simplicity and popularity, and that our paper builds upon a recent work [Bat84] which also used the ER model. Note that our choice of the ER model does not limit the generality of our approach; molecular objects, type-version generalization, instantiation, and parameterized versions

A characteristic of the ER model, as well as other semantic data models, is that two distinct logical representations of a database are used (or implied). One is a semantic representation

which expresses a database in terms of entities or objects and their relationships. The other is a mapping of a semantic representation to a more internal representation [Tsi82] (e.g., graphs or relations). For purposes of conciseness and clarity, we will map ER diagrams to relations, thereby showing how our concepts might be represented by current DBMSs.

We will present each concept by first giving its semantic definition and representation, and then providing rules by which its semantic representation can be reduced to a relational form.

3.1 Molecular Objects

Molecular objects have two levels of description: an interface and an implementation. Both levels are modeled separately using standard ER techniques. The resulting models are then integrated using the concepts of molecular aggregation and correspondence. A complete description and treatment of these concepts is given in [Bat84]. As a brief review, *molecular aggregation* treats a set of heterogeneous entities and their relationships as a single higher-level entity. It is possible for specific entities, attributes, and relationships to exist at both the interface and implementation levels. In such cases, the representations that describe them will be different at each level. *Correspondence* is a mapping which is used to show the identity of representations at different levels. Consider the following example.

Figure 7 shows the interface and an implementation of a 4-input AND gate. An ER plan of gate interfaces is shown in Figure 8a. The plan itself is quite simple; gates have zero or more pins as their external features. Pins are existence dependent on their gates and are thus represented as weak entities [Che76] or characteristic entities [Cod79].

An ER plan of gate implementations is shown in Figure 8b. Higher-level gates are composed of lower-level gates, terminals, and wires. Terminals are pins of a higher-level gates; they are used for the external referencing of inputs and outputs. Terminals are not pins of lower-level gates.

The interface and implementation plans are integrated in Figure 8a-b to form a model of a gate molecular object. The box drawn around the ER plan of Figure 8b denotes molecular aggre-

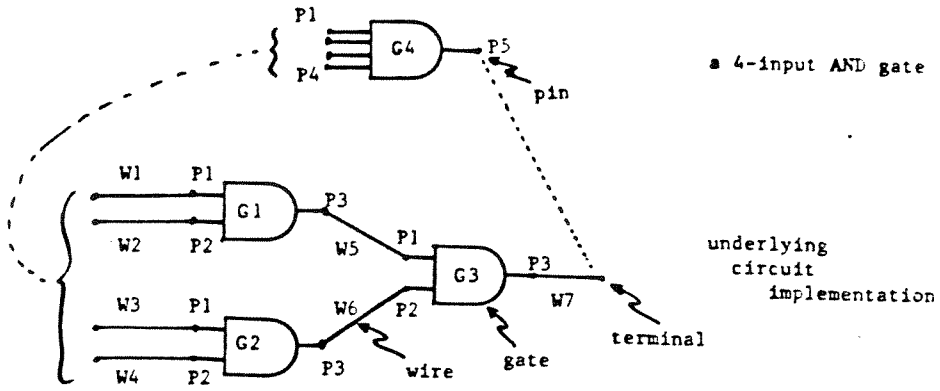


Figure 7. A Circuit Diagram of a 4-Input AND Gate

gation; it represents the aggregate of entities and relationships that define a specific gate implementation. Each aggregate is abstracted into a single, higher-level entity.

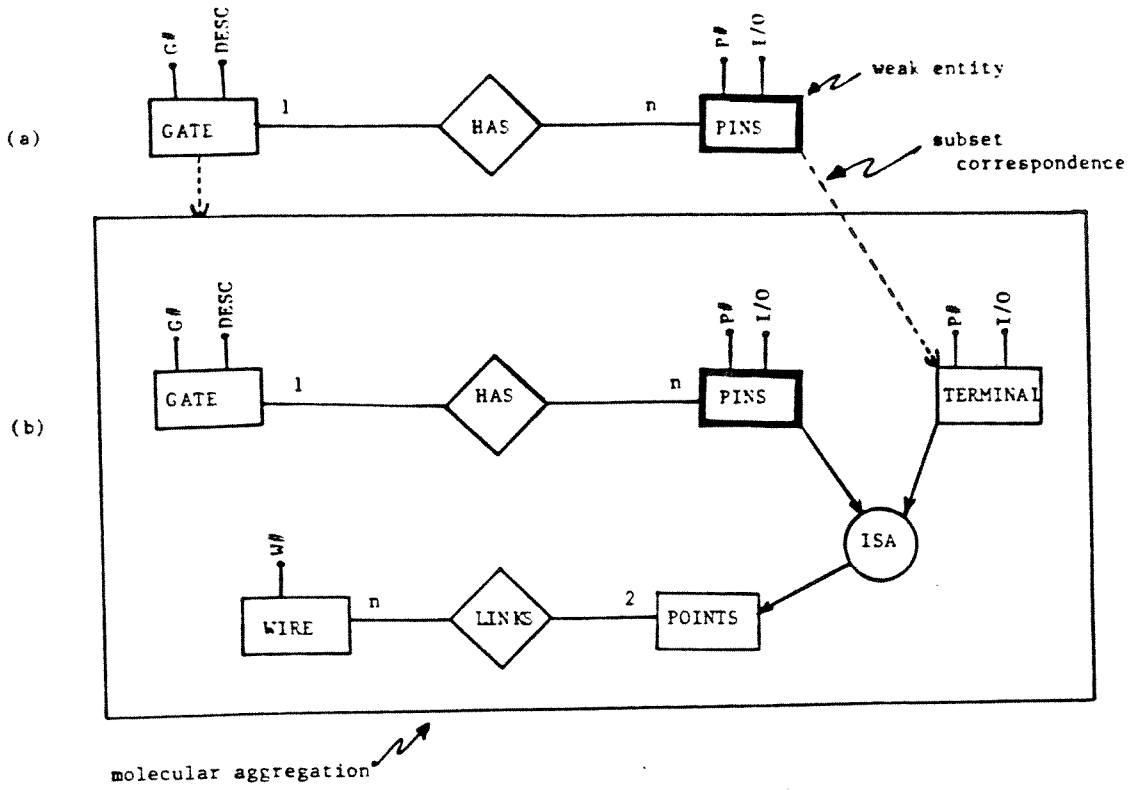
The dashed line connecting the GATE entity set in the interface with the aggregation box expresses correspondence. It means that each aggregate corresponds to precisely one GATE entity. [1] Similarly, each interface-level PIN entity corresponds to precisely one implementation-level TERMINAL entity. This too is modeled by correspondence.

Figure 8c shows the relational tables that underly the molecular object plan of Figures 8a-b, along with the tuples that define the molecular object (a 4-input AND gate) of Figure 7. The primary key of each table is underlined. Rules for reducing ER plans of molecular objects to tables are given in [Bat84].

3.2 Type-Version Generalization

Type-version generalization is the relationship between object types and their versions. Type-version generalization has two special properties. First, there is the notion of attribute inheritance; all attributes (i.e., interface properties) of an object type are inherited by its versions. Second, object types have update restrictions whereby certain attributes are *nonmodifiable*. Modifying an attribute of an object type may change the interface of the type. In such cases, a new

[1] The subsetting symbol that terminates the dashed line expresses *subset correspondence*. That is, each aggregate is associated with one GATE entity, but not every GATE entity is paired with an aggregate. Some gates, such as 2-input AND- and OR-gates, must be considered primitive and as such will be not be given explicit implementations.



(c)

GATE(G# TYPE DESC PARENT-G#)

G#	TYPE	DESC	PARENT-G#
G1	T1	2-AND	G4
G2	T1	2-AND	G4
G3	T1	2-AND	G4
G4	T2	4-AND	-

POINTS(G# P# I/O PARENT-G#)

G#	P#	I/O	PARENT-G#
G1	P1	I	G4
G1	P2	I	G4
G1	P3	O	G4
G2	P1	I	G4
G2	P2	I	G4
G2	P3	O	G4
G3	P1	I	G4
G3	P2	I	G4
G3	P3	O	G4
G4	P1	I	-
G4	P2	I	-
G4	P3	I	-
G4	P4	I	-
G4	P5	O	-

WIRE(W# START G# START P# END G# END P# PARENT G#)

W#	START G#	START P#	END G#	END P#	PARENT G#
W1	G4	P1	G1	P1	G4
W2	G4	P2	G1	P2	G4
W3	G4	P3	G2	P1	G4
W4	G4	P4	G2	P2	G4
W5	G1	P3	G3	P1	G4
W6	G2	P3	G3	P2	G4
W7	G3	P3	G4	P5	G4

Figure 8. A Model of a Gate Database

object type is created leaving the original object type unchanged. Versions of the original type are not considered versions of the new type. For example, suppose the adder-slice type of Figure 3a is modified so that it takes a pair of 2-bit inputs and produces a 2-bit sum and a carry. This

new adder-slice type is a different object type from that of Figure 3a. Moreover, versions of Figure 3a would not be versions of the new adder-slice.

An object type can have *modifiable* attributes. A count of the number of versions (implementations) of the object type is an example; it provides descriptive information and can be incremented without affecting the interface specification of the type.

The connection between molecular objects and type-version generalization is straightforward. Every object version is a molecular object. The implementation portion of a molecular object remains as is. However, the interface is factored into object type data (e.g., circuit type, input and output parameters) and object version data (e.g., creation date, name of circuit designer). That is, shared interface data is separated from version specific interface data.

Type-version generalization can be modeled in the following way. An ER plan of an object type and its features will look something like Figure 9a; there is an entity set of object types and an entity set of external features. Each object type has a type number T# as its primary key, with descriptive attributes $T_1 \cdots T_t$ (e.g., one of the T_i attributes could be a count of the number of versions of the object type). These are the attributes that form the underlying object type relation OT (Fig. 9b).

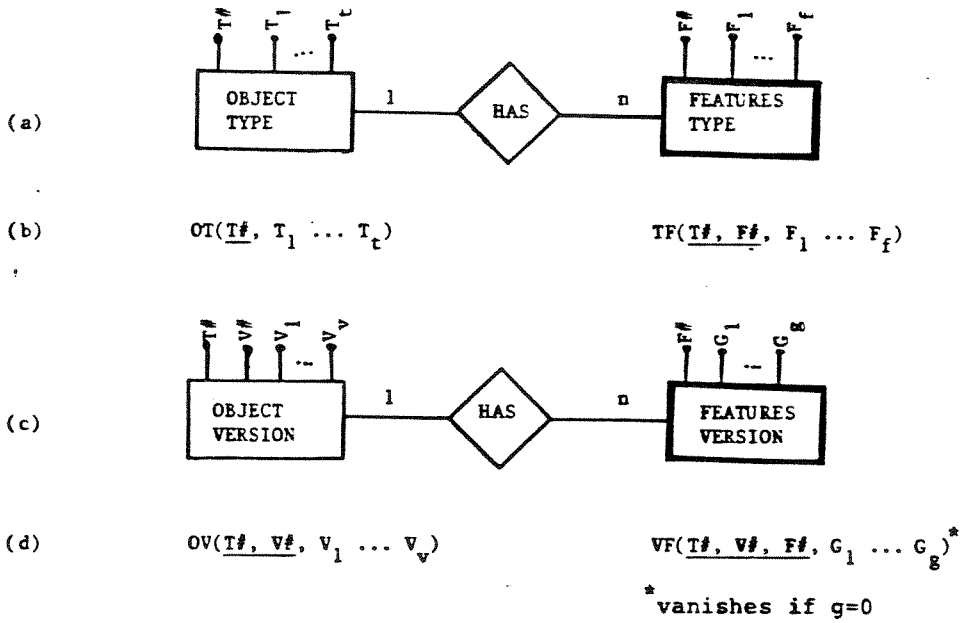


Figure 9. Modeling Concepts for Type-Version Generalization

An object type can have external features (Fig. 9a). Let FEATURE denote the entity set of such features. The attributes of FEATURE are a feature number F# and zero or more descriptive attributes $F_1 \cdots F_f$. This set of attributes, in addition to T#, forms the underlying type-feature relation TF (Fig. 9b). T# is included in TF because the primary key of a type-feature entity is (T#,F#).

An ER plan of an object version and its features is shown in Figure 9c. Each object version has a unique version number (V#) to distinguish it from other versions of the same type. It also has zero or more attributes $V_1 \cdots V_v$ that are version-specific (i.e., attributes that are not inherited from the object type). These attributes, along with T#, form the underlying object-version relation OV. The primary key of an object version is (T#,V#). Note that an object version inherits attributes $T_1 \cdots T_t$ from its object type.

The plan of Figure 9c also shows that external features of object versions may have zero or more version-specific attributes $G_1 \cdots G_g$. This set of attributes, along with T#, V#, and F#, forms the underlying version-feature relation VF. The primary key of a version-feature entity is (T#,V#,F#). Note that a version feature inherits attributes $F_1 \cdots F_f$ from its type feature.

In the special case where g=0 (there are no G attributes), relation VF reduces to three attributes (T#,V#,F#). This relation can be computed by taking the equijoin of OV and TF over T# and projecting (T#,V#,F#). Thus when g=0, relation VF is redundant and can be eliminated (Fig. 9d). VF is not redundant otherwise.

Two additional points need to be made. First, for reasons of simplicity, we do not introduce a diagrammatic notation to relate object type entity sets and object version entity sets. Instead, we adopt a naming convention where object names are followed by _TYPE or _VERSION to indicate the appropriate semantics. Second, we note that there can be any number of distinct feature types, not just one as indicated in Figure 9. If there are several, every feature type would be represented by a distinct feature entity set. Each would be modeled and reduced to tables (relations) exactly in the manner described above. If no external features are present, then the FEATURE entity sets of Figures 9a,c and FEATURE tables OF and VF are dropped.

S1 or S2 for each of the parameters of A, we can define a spectrum of unparameterized versions {A(S1,S1,S1,S1) ... A(S2,S2,S2,S2)} of an adder. Leaving any parameters unplugged (e.g., $A\,(S\,1, as_2, S\,2, S\,1)$) results in a parameterized version.

It is useful to distinguish object versions which result from plugging parameters from those that are defined using the techniques in the previous sections. We will say that an object version is *basic* if it was not formed from other versions by plugging parameters. An object version is *composite* otherwise. The adder $A\,(as_1 \cdots as_4)$ and adder-slices S1 and S2 are basic; adders $A\,(S\,1, S\,1, S\,1, S\,1)$ and $A\,(S\,1, as_2, S\,2, S\,1)$ are composite.

Although basic and composite versions are treated uniformly in our model, their relational representations are quite different. The implementation of a basic version is represented by a set of tuples that conform to a specific ER plan. The implementation of a composite version, in contrast, is represented by an encoding of its plug and socket diagram.

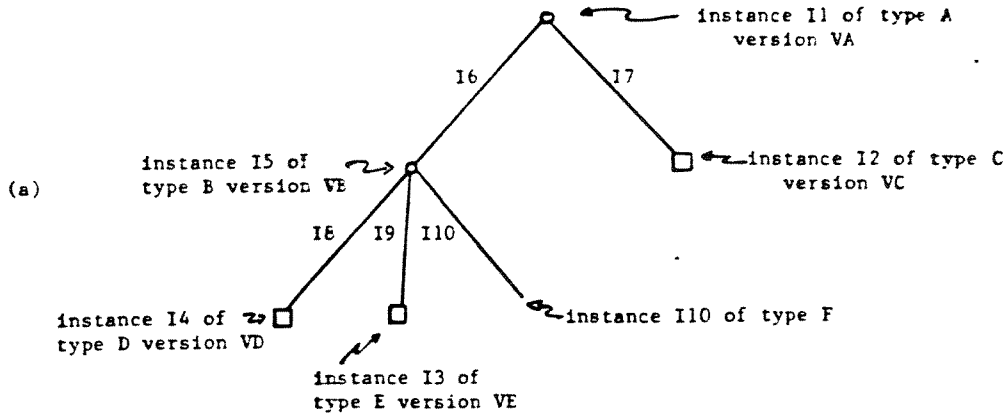
A plug and socket table PS has three groups of attributes. The first group $(T\#_{ps}, V\#_{ps})$ identifies a plug and socket diagram as the implementation of version $(T\#_{ps}, V\#_{ps})$. The second group $(T\#_n, V\#_n, I\#_n)$ distinguishes a node (version instance) in a plug and socket diagram. The third group $(T\#_s, I\#_s, V\#_p, I\#_p)$ specifies how socket $(T\#_s, I\#_s)$ of node $(T\#_n, V\#_n, I\#_n)$ is plugged with instance $I\#_p$ of version $(T\#_s, V\#_p)$. (Note that $I\#_s$ is the instantiation number of an instance of object type $T\#_s$ in version $(T\#_n, V\#_n)$). PS has a total of nine attributes. The primary key of PS is $(T\#_{ps}, V\#_{ps}, T\#_n, V\#_n, I\#_n, T\#_s, I\#_s)$ which identifies a socket in a plug and socket diagram.

Figure 16a labels the plug and socket diagram of Figure 15a with object type, object version, and instantiation numbers. Each node is given its version instance identifier (T#,V#,I#) and each arc is given its I# of an object type instantiation. Figure 16b gives table PS for Figure 16a. Note that the last row of PS describes a socket which is unplugged. This means that object

---

[2] Figure 3b actually defines an adder-slice in terms of two half-adder object type instances. If we were to be consistent, Figure 3b would be represented as an object version with two parameters of type half-adder. These parameters would be plugged with instances of the non-parameterized half-adder object version of Figure 4b.

version VA' of type A is parameterized.



(b) PS($T\#_{ps}$ $V\#_{ps}$ $T\#_s$ $V\#_s$ $I\#_s$ $T\#_o$ $I\#_o$ $V\#_p$ $I\#_p$)

$T\#_{ps}$	$V\#_{ps}$	$T\#_s$	$V\#_s$	$I\#_s$	$T\#_o$	$I\#_o$	$V\#_p$	$I\#_p$
A	VA'	A	VA	I1	B	I6	VB	I5
A	VA'	A	VA	I1	C	I7	VC	I2
A	VA'	B	VB	I5	D	I8	VD	I4
A	VA'	B	VB	I5	E	I9	VE	I3
A	VA'	B	VB	I5	F	I10	-	-

Figure 16. The Implementation Portion of Version VA' of Object Type A

4. Abstraction Relationships in Current Semantic Data Models

We are aware of two limited forms of molecular aggregation that were proposed earlier. One deals with the aggregation of a set of objects but *not* their relationships. This form of molecular aggregation has been called grouping ([Ham81]), cover aggregation ([Cod79]), and association ([Bro80], [Bro82], [Rid84]). [3] The second deals with the aggregation of treating an entire database as a single entity. This form has been called collection association ([Su83]).

There are two abstraction concepts in semantic data models that are similar to, but not the same as, type-version generalization. They are generalization (sometimes called ISA hierarchies)

[3] Smith and Smith aggregation [Smi77] (atomic aggregation) does not appear to be related to molecular aggregation. Atomic aggregation is an abstraction of a single relationship into a higher-level object; it supports *relativism*, the unification of objects and relationships in more advanced semantic data models ([Ham80], [Su83]).

and classification. The generalization concept of Smith and Smith [Smi77] takes two or more object sets (entity sets) and forms a higher-level object set by their union. Similar to type-version generalization, there is attribute inheritance. The attributes of the higher-level object set are those that are inherited by the underlying object sets. There also may be attributes which are not shared; these are the attributes that are identified with lower-level object sets. Figure 17 illustrates the idea of Smith and Smith generalization; a 'generalized' set S3 is formed by taking the union of sets S1 and S2. Note that Smith and Smith generalization is different from type-version generalization of Figure 6; a type entity set is *not* formed by the union of version entity sets.

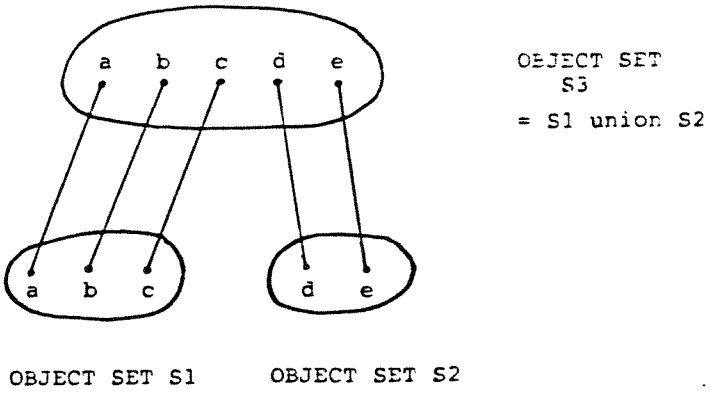


Figure 17. Smith and Smith Generalization

Classification is a simple form of abstraction in which a type is defined as a set of instances. Classification is the relationship between an object type defined in a schema and its instances that are found in a database ([Myl80a-b], [McL80], [Bor80], [Tsi82]). Type-version generalization is quite different from classification in that *both* object types and their versions are defined in a database.

We note that the term 'instantiation' poses a slight problem in terminology. Our use of the term is consistent with VLSI design methodology ([Mea80], [McL83], [EDI84], [Lor85]). However, 'instantiation' has already been used in the context of classification (described above) and programming languages to mean something quite different, i.e., an object is an 'instance-of' an object type and not a 'copy-of' another object. We will not resolve this terminological difficulty in this

paper. However, we alert readers to be aware of the different usage of 'instantiation' in other papers.

It is interesting to note that the CODASYL DDL can support a notion of type-version generalization ([Tsi77], [COD78]). Although the CODASYL model is hardly a semantic data model, it does have the facility of copying data items of owner records into virtual fields of member records (i.e., the SOURCE and RESULT declarations). Identifying owner record types with object types and member record types with version interfaces, a form of attribute inheritance is achieved.

Instantiation and parameterized versions are certainly not new to VLSI design methodology. Nor are the concepts totally new to database management systems (see [Uda84], [Wie83]). However, we do not believe that the fundamental constructs which admit instantiation and object parameterization have been recognized or formalized in the context of semantic data models. Certainly the concept of parameterized objects is not a feature of well-known models ([Che77], [Ham80], [Shi81], [Tsi82], [Su83], [Rid84]).

5. Basic Operations

The modeling concepts presented in the previous sections have led us to a first-cut set of user operations on design objects. The operations that we list below are rather primitive; they do not involve selection predicates and are procedural in nature. However, we believe that more complex operations can be expressed as sequences of these operations. We will first describe some basic operations on types, versions, and instances, and then outline a list of operations on the contents of VLSI CAD molecular objects.

6.1 Operations on Types

Let OT-set denote an object type entity set (e.g., GATE-TYPE), OT-id be an object type identifier, and OT-area be an area of main memory or secondary storage where object type data is to be stored or found. Object types can be:

- *retrieved:* READ_TYPE(OT-id, OT-area) reads object type OT-id into OT-area.

- *created:* DEFINE_TYPE(OT-set, OT-area, OT-id) takes the object type in OT-area and inserts it into entity set OT-set. OT-id, the identifier of the object type, is returned as a result. (Note that the returning of OT-id is optional if object types are assigned identifiers by users rather than by the system).

- *deleted:* DELETE_TYPE(OT-id) deletes object type OT-id from the database.

- *updated:* UPDATE_TYPE(OT-id, OT-area) replaces the old definition of object type OT-id with the object type found in OT-area. Only modifiable attributes can be changed.

6.2 Operations on Object Versions

Let OV-id be an object version identifier and OV-area be an area of main memory or secondary storage where object version data is to be stored or found. Object versions can be:

- *retrieved:* READ_VERSION(OV-id, OV-area) reads object version OV-id into OV-area.

- *created:* DEFINE_VERSION(OT-id, OV-area, OV-id) takes the object in OV-area to be a new version of object type OT-id. OV-id, the identifier of the object version, is returned as a result. (The system assigns a unique version number).

- *deleted:* DELETE_VERSION(OV-id) deletes object version OV-id from the database.

- *updated:* UPDATE_VERSION(OV-id, OV-area) replaces the old definition of object version OV-id with the object type found in OV-area. Only modifiable attributes can be changed.

6.3 Operations on Instances

Let OT-inst-id and OV-inst-id be identifiers of an object type instance and an object version instance, and let I-area be an area of main memory or secondary storage where object instance data is to be found or stored. Object instances can be:

- *retrieved:* READ_INSTANCE(OT-inst-id or OV-inst-id, I-area) reads object instance OT-inst-id or OV-inst-id into I-area.

- *created:* DEFINE_TYPE_INSTANCE(OT-id, I-area, OT-inst-id) creates an instance of object type OT-id and places it in I-area. OT-inst-id, the identifier of the instance, is returned as a result. DEFINE_VERSION_INSTANCE(OV-id, I-area, OV-inst-id) creates an instance of object versions.

- *updated:* UPDATE_INSTANCE(OT-inst-id or OV-inst-id, I-area) replaces the old definition of object instance OT-inst-id or OV-inst-id with the instance found in I-area. Only modifiable attributes can be updated.

- *deleted:* DELETE_INSTANCE(OT-inst-id or OV-inst-id) deletes the specified instance from the database.

- *plugged:* PLUG(OV-inst-id1, OT-inst-id, OV-inst-id2) plugs version instance OV-inst-id2 into socket OT-inst-id of version instance OV-inst-id1.

- *unplugged:* UNPLUG(OV-inst-id, OT-inst-id) unplugs the current version instance in socket OT-inst-id of version instance OV-inst-id. The unplugged instance is deleted.

6.4 Operations on the Contents of VLSI CAD Molecular Objects

Retrieval requests can span multiple levels of abstraction::

- given a component at level i, retrieve all of its subcomponents down to level i+n.

- given the components within a two-dimensional region at level i, retrieve all subcomponents down to level i+n.

- given a component (e.g., a flip-flop) at level i, find all instances of the same component down to level i+n.

- given a set components within a two-dimensional region at level i, find all instances of these components down to level i+n.

- given a component, find all components that are connected to its output lines.

- given a component, find all components that are connected to its input lines.

- given a component, does it have a feedback signal (i.e., is there a cycle)?

- given two components, are they connected (directly or indirectly)?
- given a component at level i, find the components at level i+n to which it is connected.

Again, the above is a provisional list of user operations. Further research is necessary to refine this list and to introduce selection predicates. A principal goal of such research is to define a nonprocedural language for querying and modifying design objects.

6. Conclusions

In this paper, we presented a framework for capturing the semantics of VLSI CAD design objects. The framework is based on the concepts of molecular aggregation, type-version generalization, instantiation, and parameterized versions. We showed that these concepts are not recognized (or are present only in restricted forms) in current semantic data models. We also presented a provisional list of user operations on VLSI CAD design objects which was consistent with our modeling approach. We believe that our framework can lead to a general semantic model of design objects.

This framework is being used as the basis of current efforts to investigate topics in version control for design databases. One topic is storage structures for versions. The problem is to minimize storage redundancy and still provide fast access to any specific version and to quickly process requests that span multiple versions. Preliminary results of our work on this topic are given in [Kim85].

Another topic is that of monitoring changes in versions and notifying affected designs. Some changes at one level of a design may impact the validity of designs at other levels [Wie82]. Further, the validity of a version v_i of a design may be affected by changes in another version v_j because v_i was derived from v_j or v_i has instances of v_j as part of its implementation. Our work on this topic is currently in progress.

Acknowledgments. We gratefully acknowledge the incisive comments of Dr. Carlo Zaniolo of M.C.C. on an earlier version of this paper.

References

[ACM80] *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling*, Pingree Park, Colorado, ACM, 1980.

[Bat84] D. Batory and A. Buchmann, 'Molecular Objects, Abstract Data Types, and Data Models: A Framework', *Proc. VLDB 1984*, 172-184.

[Bro80] M.L. Brodie, 'Data Abstraction for Designing Database-Intensive Applications', in [ACM80], 101-103.

[Bro82] M.L. Brodie, 'Association: A Database Abstraction for Semantic Modelling', *Entity-Relationship Approach to Information Modeling and Analysis*, (editor: P.P.S. Chen), ER Institute, 1981, 583-608.

[Buc84] A. Buchmann, 'Current Trends in CAD Databases,' to appear in CAD.

[Che76] P.P.S. Chen, 'The Entity-Relationship Model - Toward a Unified View of Data', *ACM Trans. Database Syst.* 1 #1 (March 1976), 9-36.

[COD78] CODASYL Data Description Language Committee, 'Data Description Language', January 1978.

[Cod79] E.F. Codd, 'Extending the Database Relational Model to Capture More Meaning', *ACM Trans. Database Syst.* 4 #4, (December 1979), 397-434.

[Dad84] P. Dadam, V. Lum, and H.D. Werner, 'Integration of Time Versions into a Relational Database System', *Proc. VLDB 1984*, 509-522.

[Dat82] C.J. Date, *An Introduction to Database Systems, Volume I*, Addison-Wesley, 1982.

[EDI84] Electronic Design Interchange Format, preliminary specification, version 0.8, 1984.

[Ege84] A. Ege, 'High Level Data Types', MCC Internal Technical Report, 1984.

[Gut82] A. Guttman and M. Stonebraker, 'Using a Relational Database Management System for Computer Aided Design Data,' *IEEE Database Engineering*, 5 #2 (June 1984), 56-60.

[Gut84] A. Guttman, 'New Features in a Relational Database System to Support Computer Aided Design', Ph.D. dissertation, Electronics Research Lab., University of California, Berkeley, 1984 (to appear).

[Ham81] M. Hammer and D. McLeod, 'Database Description with SDM: A Semantic Database Model', *ACM Trans. Database Syst.* 6 #3 (September 1981), 351-386.

[Has82] R. Haskin and R. Lorie, 'On Extending the Functions of a Relational Database System', *ACM SIGMOD 1982*, 207-212.

[Hay84] M. Haynie and C. Gohl, 'Revision Relations: Maintaining Revision History Information', *IEEE Database Engineering*, 7 #2 (June 1984), 26-34.

[IBM82] 'SQL/Data System: Concepts and Facilities', IBM Corp. Form GH24-5013-1 File No.S370/4300-50, February 1982.

[Joh83] H. Johnson, J. Schweitzer, and E. Warkentine, 'A DBMS Facility for Handling Engineering Entities', *Databases for Engineering Applications, Database Week (ACM)*, May 1983, 3-11.

[Kai82] G. Kaiser and A. Habermann, 'An Environment for System Version Control', Tech. Rep., Dept. of Computer Science, Carnegie-Mellon University, November 1982.

[Kat82a] R. Katz, 'A Database Approach for Managing VLSI Design Data', *19th Design Automation Conf.*, June 1982, 274-282.

[Kat82b] R. Katz and T. Lehman, 'Storage Structures for Versions and Alternatives', TR 479, Computer Sciences Dept., U. of Wisconsin, July 1982.

[Kim85] W. Kim and D.S. Batory, 'A Model and Storage Technique for Versions of VLSI CAD Objects', *Intl. Conf. on Foundations of Data Organization*, May 1985.

[Lor83] R. Lorie and W. Plouffe, 'Complex Objects and Their Use in Design Transactions,' *Databases for Engineering Applications, Database Week 1983 (ACM)*, May 1983, 115-121.

[Lor84] R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier, 'Supporting Complex Objects in a Relational System for Engineering Databases,' in *Query Processing in Database Systems*, (editors: W. Kim, D. Reiner, and D. Batory), Springer Verlag, 1984 (to appear).

[McL83] D. McLeod, K. Narayanaswamy, and K. Bapa Rao, 'An Approach to Information Management for CAD/VLSI Applications,' *Databases for Engineering Applications, Database Week 1983 (ACM)*, May 1983, 39-50.

[Mea80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Myl80a] J. Mylopoulos, 'A Perspective for Research on Conceptual Modelling', in [ACM80], 167-170.

[Myl80b] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong, 'A Language Facility for the Design of Interactive Database Intensive Applications', *ACM Trans. Database Syst.*, 5 #2, (June 1980), 185-207.

[Plo83] W. Plouffe, W. Kim, R. Lorie, and D. McNabb, 'Versions in an Engineering Database System', IBM Research Report: RJ4085, IBM Research, Calif., October 1983.

[Rid84] D. Ridjanovic and M.L. Brodie, 'Fundamental Concepts for Semantic Modelling of Objects', Tech. Rep., Computer Corporation of America, October 1984.

[Shi81] D. Shipman, 'The Functional Data Model and the Data Language DAPLEX', *ACM Trans. Database Syst.*, 6 #1 (March 1981), 140-173.

[Smi77] J.M. Smith and D.C.P. Smith, 'Database Abstractions: Aggregation and Generalization', *ACM Trans. Database Syst.*, 2 #2 (June 1977), 105-133.

[Sto83] M. Stonebraker, B. Rubenstein, and A. Guttman, 'Application of Abstract Data Types and Abstract Indices to CAD Databases,' *Databases for Engineering Applications, Database Week (ACM)*, May 1983, 107-113.

[Su83] S.Y.W. Su, '*SAM**: A Semantic Association Model for Corporate and Scientific-Statistical Databases', *Infor. Sci.* 29 (1983), 151-199.

[Tsi77] D. Tsichritzis and F. Lochovsky, *Data Base Management Systems*, Academic Press, 1977.

[Tsi82] D. Tsichritzis and F. Lochovsky, *Data Models*, Prentice-Hall, 1982.

[Uda84] Y. Udagawa and T. Mizoguchi, 'An Extended Relational Database System for Engineering Data Management', *IEEE Database Engineering*, 7 #2 (June 1984), 67-75.

[Wie83] G. Wiederhold, A. Beetem, and G.E. Short, 'A Database Approach to Communication in VLSI Design', *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1 #2 (April 1982), 57-63.