A TRANSACTION MODEL FOR OBJECT-ORIENTED VLSI CAD

Henry Tirri[1]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-15 June 1986

---

[1]On leave from Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Finland.

1 Introduction

The Very Large Scale Integrated circuit (VLSI) design environment is characterized by a large volume of data with complex data descriptions. Both data and descriptions of data are dynamic in nature, as well as the collection of design rules. Hence VLSI CAD is emerging as one of the most challenging application areas for database technology.

Unlike conventional database transaction processing VLSI CAD applications seem to require object-oriented accessing to design data and thus call for a different orientation than the record (tuple)-based access paradigm supported by existing database management systems. Also the traditional transaction model [Gray 78] with short-lived, non-nested transactions is not applicable to CAD environment in general [BaKK 85]. This problem is well recognized and several proposals to extend the model of transactions for design systems exist [BaKK 85], [KLMP 84], [LorP 83].

One fundamental problem that has to be addressed when developing a transaction model for VLSI CAD databases, is the nature of consistency preservation for design objects. A large portion of a design engineer's time is spent applying constraints that vary from standardized design rules to rather ill-defined rules of thumb and personal preferences [BucC 85]. It is almost impossible to specify all constraints at start-up of a VLSI CAD system. Hence there has to be an inherent mechanism to define constraints dynamically. However, allowing designers to share design objects raises the question what are the consistency constraints that these concurrent accesses to the design database have to preserve, and how this preservation should be guaranteed. The standard notion of serializability [EGLT 76] is too restrictive in VLSI CAD environment since designers tend to exchange incomplete designs.

Managing VLSI design information requires (due to the object-oriented nature) modeling constructs that are able to reflect some of the semantics of design objects. Molecular objects [BatK 85] is a database modeling construct that seems to provide a reasonable tool for developing a semantic database model for VLSI CAD objects. The key observation is that an object can be decomposed to its interface description and its implementation descriptions, which captures the nature of VLSI design objects (and also the concept of versions). The transaction model presented is strongly influenced by this type of data modeling, especially since we relate the interface description and consistency constraints.

Abstract

In this paper we address the problem of developing a transaction processing scheme in a VLSI CAD environment whose design database is based on the molecular object model. We develope a transaction model that incorporates object-oriented processing (transactions access design objects instead of tuples), hierarchical transaction structures and a notion of object consistency. To show the feasibility of our model we describe a concurrency control scheme for object-based VLSI CAD databases.

implementation is usually defined by less complex circuits and their interconnections. A circuit description is often hierarchical, since each component circuit used in the implementation has its own interface and implementation. This reflects the recursive nature of VLSI design process. As a rule, a circuit interface can have multiple implementations.

Molecular objects are objects that have an interface description and an implementation description. The implementation description of a molecular object is defined by (a heterogeneous) composition of component objects and their relationships. This abstraction is called ***molecular aggregation.*** Since a circuit interface may have several implementations, the concept of a molecular object is extended in the following way. Objects that share the same interface but have different implementations are called ***versions.*** In this case the interface description is called an ***object type.*** Intuitively an object type describes all the common features of its versions; in our case the circuit function and I/O-description.

Copies of both the object type and object versions (interface and implementation) can be generated by ***instantiation.*** Instantiation differs from duplication since it reduces redundancy; an instance of an object (version) has references to a common definition instead of a copy of this definition. Observe that using an instance of an object type creates a template in which instances of any version of the object type may be placed (this is called ***parametrized versioning*** in [BatK 85]).

2.2 General architecture of a VLSI CAD database

Following the ideas introduced in [KLMP 84] the design system considered here consists of a public database management system and private database management systems which are connected via a local area network. Public database acts as a design library, which contains predefined standard components (elementary gates etc.) and design objects that have reached a stable status. Designers' private databases contain information about design objects the designer is currently working on. In addition there exist semi-public databases which are used to exchange incomplete design objects between cooperating designers. Users of a semi-public database form a user-group that a designer can join (if authorized) or from which he can resign. Usually a user-group consists of designers involved in a common subproject of the overall design. Since we have adopted the object-oriented approach, all design objects are assumed to be represented as molecular objects.

In this paper we address the problem of developing a transaction model and a concurrency control scheme in a VLSI CAD environment whose design database is based on the molecular object model [BatK 85]. Our formal transaction model is in tradition of the general framework presented in [BaKK 85], [KorK 85]. However, the restriction we have adopted to consider especially VLSI CAD with an object-oriented data model adds some features to the transaction model, and allows us to make additional assumptions (e.g. of the operations available) than would be possible in the general case. Our study provides one answer to the question of defining a notion of consistency in a design database, namely interpreting interface descriptions as consistency constraints.

Our work is divided as follows. In Section 2 we present an intuitive model of transaction processing in VLSI CAD environment based on molecular objects. A simple formal model is presented in Section 3 which is then used for presenting a concurrency control scheme in Section 4. Finally, a brief summary and possible extensions to the ideas presented is given in Section 5.

2 Transactions in VLSI CAD environment

A typical VLSI circuit design is a top-down process that begins with a descriptive high-level specification of the design (primarily dataflow and timing graphs). The descriptive graphs are hierarchical in the sense that their components can be recursively decomposed into simpler components. There are several relationships that might be specified among the components of a high level design specification. Various constraints can be attached to the graphs; e.g the limit of a time interval and area allowed.

VLSI circuit design utilizes usually a design library, which contains components to be used in the construction of new components. It can also contain designs that are themselves under construction; subparts of a larger design or independent projects.

2.1 Modeling VLSI CAD objects with molecular objects

The description of a VLSI circuit consists of two parts: an interface description and an implementation description [McNB 83]. The interface description is the specification of the circuit function and input/output lists. The

retrieval requests address multiple levels of the design: given an object version instance retrieve all subobjects down to level i or given a set of object instances within a two-dimensional region at level i, retrieve all subobjects down to level $i+n$. Some of the retrieval requests require connection information: given an object, find all objects that are connected to its I/O-lines.

The environment briefly described above differs considerably from a conventional database application due to its object-oriented nature and hierarchically related long-lasting transactions. Consequently it is hard to see how the traditional transaction model and notion of correctness, serializability [EGLT 76] could be applied to this environment. The framework of [BaKK 85] addresses the problem of modeling hierarchical transaction relations in a CAD environment in general with a notion of consistency. However, since we are considering a specific CAD environment, VLSI CAD with an object-oriented data model, we would like to be more specific. Our contribution is the development of a transaction model that incorporates object-oriented processing (data representation, access units), hierarchical transactions in this special environment and a notion of object consistency. We also apply this model to describe a concurrency control scheme. To our knowledge this is the first attempt to develop a concurrency control scheme for an object-oriented CAD database.

3 VLSI CAD transaction model

We proceed by defining a transaction model for the object-oriented VLSI CAD environment. Our purpose is twofold: to develop a model where transactions have a hierarchical structure and access units are arbitrary complex, possibly heterogeneous objects instead of plain records. Secondly we would like to provide a framework for the construction of transaction processing mechanisms (concurrency control, recovery) for an object-oriented CAD environment.

As usual the construction of our model is based on defining notions of consistency and operations. However, at logical level our database is a collection of objects, not a collection of records or tuples. Consequently the transaction operations operate on object level (although their implementations naturally function at record level).[1] A natural consequence of this is that we also consider consistency at object level, not the primitive record level.

[1] We do not consider the possible catalog relations here.

Designers tend to subdivide large design objects (e.g. CPU) to subobjects by specifying required circuit functionalities and I/O-connections of the subobjects, i.e. the object types. Object versions can then be designed independently by designers themselves or by subcontractors [BaKK 85]. Hence we have a hierarchy of transactions (designers' actions) designing objects in various levels in parallel. In the beginning most of the objects on the higher, more abstract levels are instances of object types which gradually can be completed to object versions. During this design process design transactions manipulate design data in various ways, some of which are listed below.

A transaction can create an object type and an object version, which can be stored into the design library in the public database (checkin). This allows other transactions to create instances of this object and use them in their design process. Correspondingly, object types and versions in the public database can be deleted, assuming proper authorization, and sometimes updated (in a very restricted sense discussed in the next Section). Deletion of an object type (or version) has an effect on those design objects that have used instances of it, both in the public database and private databases.

Object types created can be stored into a semi-public database which allows creation of instances of incomplete objects. Sometimes it is useful also to allow an object version to be stored into a semi-public database. In this case the purpose is not so much to allow other transactions to make instances of an incomplete implementation but to exchange a partial product (i.e. allow it to be copied) which some other transaction could complete. This scheme might be used for example in the case where some of the designers are "specialists". Instances of object types in semi-public databases should not be used in object versions that are checked into the public database before the creation of the object type is confirmed (i.e. the object type is stored into the public database) since these interface specifications tend to be unstable.

During a design of an object version instances of existing object types and versions can be created, deleted and updated (although update is again allowed only in a very restricted sense). It is also sometimes useful to copy an object instead of creating an instance of it, for example when a new object version is to be created that is based on a previous design.

In addition to the possibility of creating and deleting objects designers also need information about the contents of design objects. Typically

Definition 3 *Assume that χ is the interconnection function (defined by wire objects) of object version γ. Type δ of an object version γ is $\chi(\Theta \cup I)$ where $\Theta = \{\theta_i\}$ is the set of objects in γ and $I = \{I_j\}$ is the set of object instances in γ $(i, j \geq 0)$.*

A type of an object version is its specification based on the specifications of its components and their interconnections. With this notion we are able to define consistency in our transaction model.

Definition 4 *An object θ is consistent if for all $\gamma \in \Gamma$ we have $\delta(\gamma) \Rightarrow \tau_\theta$. In addition an atomic object is consistent and $\delta(\varphi)$ always implies any τ_θ. An instance I is consistent if the corresponding object $(\tau, \Gamma), \Gamma = \{\gamma\}$ is consistent.*[3]

Thus a design object is consistent when the implementations satisfy specifications. Note the difference between an object having a null version φ and having no versions at all. In the latter case the object is atomic and does not require an implementation, in the former case we have a design object whose specification has been created, but no implementation exists for this specification. The definition of instance consistency is very flexible. It allows an instance I to be consistent although the object it is derived from has an incomplete version γ', naturally assuming that $\gamma \neq \gamma'$.

Definition 5 *A state S of a database $\mathcal{D}$ is a pair (S_θ, S_I) where S_θ is a set of objects and S_I is a set of instances. State S is consistent if for all $\theta \in S_\theta, I \in S_I$ object θ and instance I are consistent.*[4]

The basic operations that designers use operate on object level. These object operations are implemented by atomic collections of lower level operations (e.g. record level). However, for the purposes of our transaction model it is sufficient to consider object operations as atomic. Ensuring atomicity of object operations is an interesting topic, but is outside the scope of this paper.

[3] Note that a specification τ with a null version φ is always consistent since there exists no contradiction between the specification and the implementation.

[4] Our simple definition of a database reflects the scope of this paper, which is limited to consideration of design objects only. A real design database has also homogeneous, relational data about the design data. However, from the transaction processing point of view this data can be managed with traditional methods and is not of interest here.

Therefore we depart from the traditional approach of defining a database state as a set of record values and consistent states being states that satisfy a given predicate C, and introduce the notion of *object consistency* on which our transaction model is based. This approach also allows us to reflect the dynamic nature of defining constraints during a design process, we do not require that all the object constraints are specified at start-up time.

Similarly our definition of a transaction is biased towards maintaining object consistency. Intuitively our transactions are partially ordered sets of transactions or atomic operations that create an object version (circuit implementation) which satisfies given constraints.

In the following we use the term database to mean the collection of all the databases in the environment. In the abstract level of our model it is not necessary to make a distinction between the different database types, although this distinction becomes important in Section 4, where we discuss implementing concurrency control mechanisms for the VLSI CAD environment.

Definition 1 *An object θ is a pair (τ, Γ) where τ is an object type and Γ is a set of object versions γ. An object version γ is a set of objects. An object version that has no components is denoted by φ and it is called a null version. An object is an atomic object if it does not have an object version (i.e. Γ is empty). We explicitly exclude recursive objects.*

The formal definition of an object is intended to capture the nature of a VLSI design object. Object type τ is the interface specification of the circuit. This specification usually takes form of a functional specification (including timing) and I/O-interface specification (number of pins etc.). Object version γ is the implementation of the design object using usually less complex circuits. An atomic object is a standard component available in the design library. Since a circuit should not be used to implement itself recursion is not allowed.[2]

Definition 2 *An object instance I is a 3-tuple (τ, γ, id) where τ is an object type, γ is an object version and id is a unique instance name. If $\gamma = \varphi$ I is an object type instance which is denoted by I_τ.*

[2]Following the classification in [BatB 84] our design objects fall into the category of non-recursive, disjoint/non-disjoint objects.

pins) of the lower level circuits, since this information belongs to the details of the implementation which the specification is intended to hide.

Our transactions are arbitrarily nested hierarchical structures. To describe a transaction in terms of atomic operations we use the notion of *transaction closure* introduced in [BaKK 85].

Definition 8 *The closure* t^* *of a transaction* $t = (T, O, \prec, C)$ *is a 3-tuple* $(O^*, \prec^*, C)$ *where* $O^* = O \cup \{o^* \mid (t, o, \prec, c) \epsilon T\}$ *and* $\prec^*$ *is defined as follows: assume that* $r, s \epsilon O^*$. *We have* $r \prec^* s$ *if one of the following conditions holds:*

- $r, s \epsilon O$ *and* $r \prec s$,
- *there exists a transaction* $t_i = (T_i, O_i, \prec_i, C_i) \epsilon T$ *such that* $r, s \epsilon O_i^*$ *and* $r \prec^* s$,
- *there exists a transaction pair* $t_i = (T_i, O_i, \prec_i, C_i)$, $t_j = (T_j, O_j, \prec_j, C_j) \epsilon T$ *such that* $r \epsilon O_i^*, s \epsilon O_j^*$ *and* $t_i \prec t_j$ $(i \neq j)$.

Transaction closure flattens the transaction hierarchy, but preserves the ordering constraints of the operations. Observe that in general the ordering $\prec^*$ remains partial. This brings us to an important difference between the traditional transaction model and the nested models such as ours. Traditionally transactions are assumed to be units of global consistency, hence if a transaction is executed alone it maps a consistent database state to a consistent state. Nested transactions usually represent a set of transactions that can be executed concurrently assuming that the imposed ordering constraints are obeyed. Requiring a nested transaction to be a unit of global consistency would be equivalent to requiring a set of traditional transactions to coordinate their execution without any concurrency control mechanism. However, our transactions are units of consistency with respect to the nesting level; i.e. a transaction preserves consistency if the subtransactions preserve consistency.

When a transaction is executed system forces additional ordering among the operations. In a distributed workstation environment several operations can be executed in parallel, but the operations using a same executing agent are serialized. Hence for the definition of a *transaction execution* we attach an executing *agent* to each operation.

Definition 6 *A (database) operation o is a mapping $o : S \rightarrow S$.*

Although our model does not restrict the nature of object operations, for simplicity we assume in the sequel that the operations allowed fall into one of the following general categories: create/delete object types, create/delete object versions, create/delete instances, retrieve information about the contents of an object/instance, retrieve information about the environment of an object/instance, update statistical or identification information in object/instances. We also feel that these categories are representative of the operations required in object-oriented VLSI CAD.

There is an interesting difference between the operations in an object-oriented design environment and traditional database management systems. The design transactions tend to create new versions rather than update old ones; construction of a circuit implementation is done in the private workspace of the designer's workstation and stored to the database after the design is completed (or partially completed). Hence in our model completing an incomplete version is not seen as updating, but creating a new version. This method allows a natural way of storing a design history which makes it possible to use earlier versions to test alternative design decisions. When versions become obsolete they are deleted. Updating objects is allowed only in a very restricted way; only *modifiable* attributes [BatK 85] such as number of versions of an object type can be updated. The concept of multiple versions of data exists also in traditional database applications [PapK 84],[Reed 78], but it is usually introduced to enhance the transaction processing scheme (concurrency control and recovery). In the VLSI design environment these versions are used in the design process, hence very little additional overhead is involved in allowing the transaction processing scheme to use a version strategy.

Definition 7 *A transaction t is a 4-tuple $(T, O, \prec, C)$ where T is a set of transactions, O is a set of operations, $\prec$ is a partial order on $T \cup O$ and C is a consistency constraint. Every transaction is of finite depth.*

This notion of a transaction is influenced by the general definition in [BaKK 85]. A transaction can be seen as a set of possible mappings from the database state S to S. However, in our case the consistency constraints of the subtransactions in T are not necessarily implied by C. This is due to the nature of our consistency constraints. The specifications of higher level circuits do not describe for example the I/O-interface (number of input pins and output

to apply the notions of our formal model and describe a concurrency control protocol for an object-oriented VLSI design database. As will be seen, there are interesting similarities and differences between our approach and traditional approaches for constructing concurrency control protocols in traditional (record-based) databases. To be able to present our protocol, we have to be more specific in our description of the VLSI design environment.

4.1 Databases and workareas

For our purposes it is sufficient to distinguish the following databases in a VLSI design system:

- public database ($PUDB$)
- semi-public databases ($SPUDB$)
- private databases ($PRDB$)
- design constraint databases (CDB)

The *public database* is a project-wide database which is a collection of stable design objects[5] from which new object copies (instances) can be generated. For simplicity it is also assumed that the standard design library is included in this public database. Object insertion to the public database as well as object deletion requires proper authorization.

A *semi-public database* is used to store design objects that have not yet reached a stable status, but could be useful to more than just one designer, i.e. they are used to exchange design objects. To check out information from a semi-public database a designer has to belong to the user-group of the database. These user groups are formed dynamically to reflect current organization of the project.

A *private database* is a designer's local database that has objects extracted from the public database and semi-public databases. The objects extracted are determined by the type of the design task, i.e. the private database is intended to contain those objects that a designer uses in the

[5]In this Section we use term object loosely to denote a basic data unit such as an object type, object version or object type instance. When we want to stress that the object is of certain type we use the appropriate term.

Definition 9 *Let $\mathcal{A}$ be a set of agents and $A_t : O^* \rightarrow \mathcal{A}$ a mapping that relates an operation o and the executing agent $A_t(o)$. Then an execution e_t of a transaction $t = (T, O, \prec, C)$ is a pair $(O^*, \prec')$ where $\prec'$ is a partial order defined as follows: assume that $r, s \epsilon O^*$. Then the following conditions hold:*

- $r \prec^* s \Rightarrow r \prec' s$,
- *for each pair of operations (r, s) such that $A_t(r) = A_t(s)$ we have either $r \prec' s$ or $s \prec' r$.*

Definition 10 *The effect of an execution e_t is a finite collection $F = \{f_i\}(i \geq 1)$ where $f_i : S \rightarrow f_i(S)$ is defined as follows: let $o_1, o_2, ..., o_n$ be the ordered list of operations that have the same agent $A_t(o)$. Then f_i is the composition $o_1 \circ o_2 \circ ... \circ o_n$. The resulting state from an execution e_t is $S' = \cup f_i(S)$. An execution e_t is consistency preserving if S is consistent $\Rightarrow S' = \cup f_i(S)$ is consistent.*

Analogously to the traditional transaction model, allowing arbitrary transaction executions also introduces the possibility of creating inconsistencies. Hence the set of consistency preserving executions E_t describes those executions that are "correct" in the sense that they leave the database in a consistent state. Since incorrect executions should not be allowed the system has to ensure that only correct executions are allowed. A *protocol* is a set of rules that describes a subset of possible executions.

Definition 11 *A protocol is a predicate P on a set of transaction executions E_t. If for $e \epsilon E_t$ $P(e)$ is true, we say that e is legal under P. A protocol P is consistency preserving if for all $e \epsilon E_t$ $P(e) \Rightarrow e$ is consistency preserving.*

Ideally we would like to have a protocol that defines the exact set of consistent executions. However, in practice protocols have to be restricted to approximate this set, since testing an execution for consistency preserving is an intractable problem in general, even in the case of serializability [Papa 79] which is a stricter notion than our consistency preserving.

4 A concurrency control scheme for VLSI CAD

The transaction model defined above is very general and allows a wide variety of transaction implementation schemes. In this Section our purpose is

about the contents of an object can be viewed in many ways: retrieve all subobjects down to level i, given a set of objects within a two-dimensional region at level i find all instances of these objects down to level $i + n$ etc. Developing an efficient implementation to handle these retrieval requests is a research issue by itself. From concurrency control point of view retrievals are problematic only when a consistent view is required in the presence of concurrent insertions and deletions.

4.3 Transactions

Our model in Section 3 allows a transaction to invoke subtransactions that are executed in parallel with the parent transactions. This allows modeling of the situation, where a designer wants to split the design process to two or more design processes which can be progressed in parallel by the designer itself[6] or by some other designers (client/subcontractor -relationship of [BaKK 85]). The structure of a typical transaction is shown in Figure 1. The purpose of the initial checkout phase is to create a local database against which retrieval requests can be performed instead of retrieving related data from the public database. After the initial checkout transactions enter a design phase, where new designs are constructed in the work area and stored to the private database. Also from time to time additional checkouts from the public database are possible (adding recent designs). At a design phase a transaction can originate subtransactions that operate on the same private database as the parent transaction ("tasks") or on another private database ("subcontraction"). In both cases the subtransactions can outlive the design phase. In a checkin phase a transaction can make object types or object versions partially public by inserting them to a semi-public database or make them stable by inserting them to the public database. It is not required that all the insertions have to be done at the end of the transaction, hence there may be several checkin phases in a transaction.

4.4 Concurrency control protocol for object consistency

The notion of database consistency introduced in Section 3 is based on consistent objects, hence to maintain the database consistency the concurrency control protocol has to prevent transactions from introducing inconsistencies by creating inconsistent objects. In addition it has to be able to enforce

[6] This is easily realizable through the windowing support of modern workstations.

Checkout phase

- *read object types and versions from PUDB and SPUDBs into PRDB*

Design phase

- *create instances of object types and versions in PRDB, PUDB and SPUDBs*
- *retrieve information about the contents of objects in PRDB, PUDB and SPUDBs*
- *delete objects from PRDB*
- *start a new subtransaction (i.e. open a new window) etc.*

Checkin phase

- *insert object types and versions to PUDB and SPUDBs into PRDB*

Design phase

- *create instances of object types and versions in PRDB, PUDB and SPUDBs*
-
-

    -
    -
    -
    -

Checkin phase

- *insert object types and versions to PUDB and SPUDBs to PRDB*

Figure 1: The structure of a possible transaction in a VLSI CAD environment

consistency even if deletion of an object may have an effect on the consistency of other objects. Both of these requirements differ considerably from what is usually expected from a concurrency control protocol in a traditional application area. In VLSI CAD environment the protocol has to cope with inserts and deletes and dynamic checking of transaction consistency instead of writes and static consistency checking in traditional database applications. We proceed by describing the overall concurrency control scheme for maintaining object consistency. A more detailed discussion on the implementation of this scheme will appear in a sequel to this paper.

The implementation of the *object consistency protocol* (OCP) consists of five algorithms:

- lock manager ($PULM$) for $PUDB$
- lock manager ($SPULM$) for $SPUDB$s
- lock manager ($PRLM$) for $PRDB$s
- object consistency checker (OCC)
- object disconnector (OD)

Each database has a lock manager of its own, object consistency checkers are local to each workstation (private database management system) and object disconnectors are included in the transaction management system of semi-public databases and the public database.

Lock managers provide a mutual exclusion mechanism which guarantees atomicity of object insertion, object deletion and retrieval operations. The objects in the databases are typically non-disjoint, hence to prevent the creation of "orphan objects" in a deletion operation all the instances of the object have to be locked before the deletion can take place. Parallel to the search for all the instances some other transaction may try to insert a new instance. Consequently in this environment the problem of phantoms [EGLT 76] is realistic. Solving the problem with physical locking methods is very cumbersome, hence the lock managers use logical level locking. The method we use is related to *precision locks* introduced in [JoBB 81].

It is assumed that in order to delete an object the database system has to first find out if the object really exists. Therefore the whole database is first searched to find the object to be deleted before an acknowledgement is sent

to the transaction. Each retrieval operation r has an associated descriptive predicate P_r. Deletion operation carries the name of the object to be deleted (n_d) and insertion operation the object to be inserted (θ_i). As the lock manager receives operations it extracts the predicates, object names and objects and tries to enter them into corresponding lists (L_r, L_d, L_i). Before P_r is inserted in the list it is checked against L_d and L_i to see if there exists an object θ such that $P_r(\theta) = true$ (or $P_r(n_d) = true$). If so then the read space of request r is being modified and r is blocked until the insert or deletion operation is finished and the corresponding item is removed from the list. Similarly an insert operation is checked against retrieval and delete lists and a delete operation against retrieval and insert lists. It should be observed that there is no retrieval vs retrieval check, insert vs insert check or delete vs delete check, hence these operations can be run in parallel. The lock manager algorithms are essentially the same in all the databases $(PUDB, SPUDB$ and $PRDB)$, but in the semi-public databases and in the public database lock managers and object disconnectors are integrated to form one module.

Object consistency checker algorithm is invoked when an object version (or a new object type with an object version) insertion is attempted to a semi-public database or the public database. We do not require that a private database is consistent; in fact it is inconsistent most of the time during a design process, since it contains incomplete versions. Note that the lock managers do not maintain object consistency, they only guarantee that the basic operations are performed correctly in the presence of other simultaneously executed operations. The purpose of the object consistency checker is to prevent designers from making inconsistent objects public. Observe that an object type τ with a null version φ is always consistent, hence a designer can introduce new objects under design by making an object type public.

An OCC algorithm is essentially a special purpose theorem-prover which uses the specifications of the subobjects and the interconnection function defined by the wire objects to construct the type $\delta(\gamma)$ of the object version γ. This type is checked against the object type τ. If we have $\delta(\gamma) \Rightarrow \tau$ insertion is allowed, otherwise operation is aborted and diagnostic information is returned to the transaction. This process can also be semiautomatic. In addition OCC also performs constraint checking against the design rules in the appropriate constraint databases. Although OCC is a rather complicated module, it is only a variant of the design verification modules already used in the engineering community.

Finally, *an object disconnector* is the algorithm that controls the complicated situation when an object used by other transactions is deleted. This situation arises for example when a designer decides to choose an alternative implementation for the object and wants to delete an object version that he has made public by inserting it to a semi-public database. When a transaction checks out an object from a semi-public database it becomes a child transaction of the creator transaction with respect to this object.

We denote the creator transaction by $t_c(\theta)$ for each object θ. The child relation is maintained in the database where the object is stored. When a deletion of an object θ occurs all the child transactions t of the creator transaction $t_c(\theta)$ are informed by a deletion message which, after displaying an appropriate message to the designer, generates a delete operation for the object instance (or copy).

The child transactions respond with an acknowledgement when the designer has acknowledged seeing the message and the object has been deleted. After receiving acknowledgements from all the child transactions the actual deletion operation can be performed. Although this seems to be a rather time-consuming procedure it is due to the nature of the problem; objects can not be allowed to simply "vanish" and leave holes in the design. Also replacing an object involves usually design decisions and cannot be done automatically.

5 Conclusions

In this paper we have proposed a transaction model for object-oriented VLSI CAD with an intuitive description of the environment itself. We have chosen molecular objects as our data model, since it seems to reflect the object-oriented nature of CAD designs in general and allows us to depart from the traditional view of database consistency. However, the notion of object consistency is very general and assumes only that an object consists of two parts: specification and implementation. Hence it can be applied to any object-based design environment.

We applied our model to present a concurrency control scheme in VLSI CAD environment. This scheme is one possible implementation of preserving object consistency, alternative schemes can be constructed. This work is a first attempt to provide a suitable framework for developing transaction processing mechanisms for object oriented VLSI CAD. Therefore we

have deliberately tried to keep the model as simple as possible, to be able to set the stage for further expansions. In addition to concurrency control the transaction processing architecture requires methods to implement object queries efficiently and a recovery mechanism, issues that are topics for further research.

Acknowledgements

We would like to thank Hank Korth for helpful discussions and suggestions.

6 References

[BaKK 85 / Bancilhon, F., W.Kim and H.Korth, A model of CAD transactions. Proceedings of the 11th International Conference on Very Large Databases, 1985, 25-33.

[BatB 84 / Batory,D. and A.Buchmann, Molecular objects, abstract data types, and data models: a framework. Proceedings of the 10th International Conference on Very Large Data Bases, 1984, 172-184.

[BatK 85 / Batory,D. and W.Kim, Modeling concepts for VLSI CAD objects. ACM Transactions on Database Systems, 9:3 (1985), 322-346.

[BucC 85 / Buchmann,A. and C. Perez de Celis, An architecture and data model for CAD databases. Proceedings of the 11th International Conference on Very Large Data Bases, 1985, 105-114.

[EGLT 76 / Eswaran,K., J.Gray, R.Lorie and T.Traiger, The notion of consistency and predicate locks in a database system. Communications of the ACM, 19:11 (1976), 624-633.

[Gray 78 / Gray,J., Notes on database operating systems. IBM Research Report: RJ2188, IBM Research, 1978.

[JoBB 81 / Jordan,J., J.Banerjee and R.Batman, Precision locks, ACM Conference on Management of data (SIGMOD'81), 1981, 143-147.

[KLMP 84 / Kim,W., R.Lorie, D.McNabb and W.Plouffle. A transaction mechanism for engineering design databases, Proceedings of the 10th International Conference on Very Large Data Bases, 1984.

[KorK 85 / Korth,H. and W.Kim, A concurrency control scheme for CAD transactions. Technical Report TR-85-??, Dept. of Computer Sciences, University of Texas at Austin, 1985.

[LorP 83 / Lorie,R. and W.Plouffle. Complex objects and their use in design transactions. Proceedings of Databases for Engineering Applications, Database Week 1983 (ACM), 115-121.

[McNB 83 /McLeod,D., K.Narayanaswamy and K.Baba Rao, An approach to information management for CAD/VLSI applications. Databases for Engineering Applications, Database Week 1983 (ACM), 39-50.

[Papa 79 / Papadimitriou,C., The serializability of concurrent database updates. Journal of the ACM, 26:4 (1979), 631-653.

[PapK 84 / Papadimitriou,C. and P.Kanellakis, On concurrency control by multiple versions. ACM Transactions on Database Systems, 9:1 (1984), 89-99.

[Reed 78 / Reed,D., Naming and synchronization in a dezentralized computer system. PhD dissertation, MIT, Dept. of EECS, 1978.