Transaction Control For Cooperative Applications

Sridhar Ramaswamy and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-25
June 5, 1991

Transaction Control For Cooperative Applications*

Sridhar Ramaswamy and Stanley B. Zdonik
Brown University, Providence, RI 02912

June 5, 1991

Abstract

We analyze existing models of design transaction control systems. We use the following criteria: flexibility in specifying concurrency control, computational complexity of checking consistency, computational complexity of concurrency control, support for abstraction, and support for recovery. Based on this analysis, we propose a new way of doing concurrency control and recovery. We then discuss the advantages and limitations of this approach and outline open problems in this area.

*Support for this research is provided in part by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under Contract N00014-83-K-0146, by Apple Computer, Inc., and by US West.

1 The Problem

Serializability has long been held as the notion of correctness of transaction execution in databases. This is because transactions that run in most "traditional" databases (banking systems and airline reservation systems are good examples of such databases) satisfy certain criteria implicitly.

The most important of these is that the duration of transactions is short. Technically, we say that the average duration of such a transaction is a very small fraction of the Mean Time Between Failure of the underlying computer system. This turns out to have far-reaching consequences for concurrency control. Also, most transactions run independently of one another. Both these imply that the cost of aborting a transaction and rerunning it is not very high. If the assumption that all transactions are "correct" in some sense (i.e., we don't assume that a transaction is malicious) is made, then the notion of serializability serves as a good correctness criterion for doing transaction concurrency control. If a transaction aborts in the middle of its execution, we simply "wipe out" its effects using a log. That is, transactions are thought of as being atomic. Either they run completely, or they don't run at all (because the effects of partial transactions are removed).

The use of persistent data has now extended to applications that have properties different from the ones listed above. One major area is that of design transactions, which include applications like CAD/CASE tools, office information systems and interactive programming environments. They manipulate data that is often huge and very complex and intricately interrelated by numerous consistency constraints. This is very different from that of databases supporting banking systems which have relatively uniform data that are not very interrelated.

We will give an example in computer-aided design. Let us suppose that a car company wants to design a new car. They subdivide the design into three major parts, namely, designing the engine, the body, and the interior. Now, designing the engine is once again divided into designing the actual engine—given the constraints about the body and the interior—and designing the transmission. And similar natural subdivisions occur in the designing of the body and the interior also. What is to be noted here is that the natural subdivisions do not separate the problem completely. Numerous constraints will still persist across the subdivisions.

We now can make some observations about the properties of design transactions and the impact they have on the transaction control system of the database that they run on. Firstly, they manipulate data that is often complex, and intricately connected by numerous consistency constraints. Therefore, we would expect the underlying database system to be object-oriented.

Secondly, transactions tend to be very long. Designing a car can take a significant amount of time. (In our technical characterization, we can say that the average duration of such a transaction is a significant fraction of the Mean Time Between Failure of the underlying computer system.) Previously, failure of transactions and machines was handled by "wiping out" the effects of unfinished transactions. Here, the long duration of the transactions implies that there is a heavy price to be paid for treating these transactions as atomic. A significant amount of work has to be redone again and this may not at all be possible some times. Failure of transactions and machines has to be handled differently. This means that the

recovery mechanism has to be powerful and versatile enough to partially undo transactions while preserving the consistency of the database.

Thirdly, we saw from the example that tasks divide and subdivide themselves into natural hierarchies. Therefore, we would expect the control system to support nesting of transactions. Further, such nesting would also help increase concurrency while helping to keep recovery under control.

Serializability is also no longer sufficient. Since transactions doing design may manipulate the same data many times before any of them finish, it may often not be possible to serialize them. What the designers of the transaction control system could do is to let conditions be specified about the nature of interaction between various transactions. (Serializability could be one of them. In cases where cooperation is not necessary, this could still be very useful.) This mechanism should be flexible enough to allow the designers to easily specify their constraints. It should be also be possible to enforce these specifications efficiently while doing concurrency control. As we shall see, the implenting the specifications dominates the cost of doing concurrency control. And in spite of all this, we should still be able to manage recovery as automatically as possible.

In this section, we have stated the requirements of a transaction control system supporting a design environment. In the next section, we give a brief overview of related research in this area. In section 3, we examine further the requirements and study their impact on the model that we would like to have. In this section, we also mention how currently proposed schemes compare with the results of our analysis. In section 4, we examine specifications for correctness criteria in detail. In section 5, we give the salient points of our new model and its advantages. We conclude with the limitations of the currently proposed model and mention what we think will be good extensions to this work.

2 Related Research

In this section, we describe research that has been done in hierarchical organization of transactions. We also examine some models where the synchronization mechanisms do not use serializability as the correctness criterion. Instead, these models use some constraint-based definition of correctness, or use semantic information about the data or the transactions (possibly supplied by the user) to increase concurrency.

In the *Nested Transaction Model* ([Moss85]), transactions can contain complex operations as their suboperations. Normally, we view a transaction as a totally (chronologically) ordered sequence of operations. In the nested transaction model, a transaction can contain other subtransactions that run in parallel with the parent and other children of the transaction. These subtransactions can in turn have children of their own. This nesting can be extended arbitrarily, producing a transaction tree. [Moss85] suggested a version based recovery method for nested transactions, where essentially each subtransaction saves the pre-images of the entities that it wants to operate on. Nested transactions embody important ideas for the model that we want to have, namely, hierarchical organization of transaction and support for abstraction. But they still use serializability as their synchronization mechanism for siblings. For cooperative applications, we need more.

[KKB85] also suggest a hierarchical organization of transactions and defines a new correctness criterion called *Predicatewise Serializability*. The basic idea is that if the database consistency constraint is expressed in Conjunctive Normal Form, we can maintain this constraint be enforcing serializability only with respect to items that share a disjunctive clause (The paper suggests the enforcement of this constraint by using two-phase locking and calls the synchronization mechanism "Predicatewise Two Phase Locking"). This gives us increased concurrency because the serializable schedules for the different clauses do not have to agree. [KoSp88] formally combines several scheduling mechanisms known (including the one above) and defines a new class called *Conflict Predicate Correct*. Conflict Predicate Correctness, while being efficiently realizable in practice, strictly contains all Conflict Serializable (for a definition, see [BHG87]) schedules. [KLS90] extend the idea of predicate-wise serializability to *entity-wise serializability* where serializability is maintained with respect to each entity rather than a set of entities.

Sagas, proposed by [GaSa87], provides a mechanism for a two level break up of long-lived transactions. A *saga* is defined as a sequence of atomic transactions $\{T_1,T_2,\ldots,T_n\}$ with compensating transactions $\{C_1,C_2,\ldots,C_n\}$ for each transaction in the saga. A saga execution is complete when all the atomic transactions are executed in sequence. The effects of a sequence $\{T_i,T_{i+1},\ldots,T_j\}$ can be undone by executing the sequence $\{C_j,C_{j-1},\ldots,C_i\}$. Thus, when a particular transaction fails, the system can execute a set of compensating transactions and continue. Atomicity is preserved in the transaction level and not at the saga level. This paper assumes that it is possible to write long-lived transactions as a set of short steps with counter-steps.

[Lync83] describes a scheme for increasing concurrency in transaction executions. In this scheme, each transaction is split into a number of steps. These steps are then described by a sequence of k equivalence classes, each of which is a *refinement* of the previous one. The relations between transactions are described similarly. The level of interleaving of two transactions is controlled by the lowest level at which they are in the same equivalence class.

The disadvantage of this scheme is that the set of transactions is fixed. Each new addition will have to have its behavior compared with the entire set of transactions permissible.

[Skar89] defines a model for cooperative transactions in the context of an object-oriented database. Concurrency Control is achieved by defining allowable sequences of operations, called *patterns* and forbidden sequences of operations, called *conflicts*. They are defined over the methods in the database.

[NFSZ90] defines a similar model with a simplified set of operations, namely read and writes. It uses a dynamic hierarchy of transactions and also provides a mechanism to handle failures. We use some of [Skar89]'s and [NFSZ90]'s work here and will refer to some of their details in the next section.

3 Development of a Transaction Model

In this section, we examine the requirements of design transations in more detail and their impact on the transaction model.

3.1 Hierarchical Organization of Transactions

There are compelling reasons to have transactions organized in a hierarchical fashion for design transactions. First of all, such a nesting reflects a natural breakdown of work common in most design environments. In other words, tasks can be abstracted away in subtransactions, facilitating overall management. Secondly, if the concurrency control mechanism is flexible enough (and we will be soon be examining how we can make it very flexible), we gain a lot of concurrency from siblings executing in parallel, as opposed to straight-line sequencing of a transaction. Thirdly, nested transactions help keep recovery under control. When a subtransaction fails, we only have to undo its effects and this effectively shields the parent from having a lot of its work undone. Thus aborts have a localized effect unlike in the case of a flat transaction model.

It also makes sense to allow arbitrary nesting, because it is not possible to determine the maximum nesting a problem can have a priori. Also, we cannot determine the clustering of transactions before hand. A fixed hierarchy is somewhat artificial. We would like to allow the hierarchy to have members added to it (or removed from it) as the need arises. This need may arise after some parts of a transaction have already some work. In this sense, the hierarchy that we support should be dynamic. The root transaction always exists on startup of the database and maintains it. Children of the root correspond to unrelated tasks and are separated from each other.

We use the terminology of [Skar89] and say that a *Cooperating Transaction* (CT) is a program that issues a stream of operations to the underlying database. It is a leaf in the transaction hierarchy. A *Transaction Group* (TG) corresponds to an internal node in the hierarchy and is a collection of CT's or other TG's. This collection of children cooperate to perform a specific task. We have to control the way members of the hierarchy interact. We will have to allow this control to be specified by the user because pre-specified control mechanisms like serializability among siblings is not sufficient. Further this specification will reflect the semantics of the application as well as the data.

Let us consider the example that we mentioned in the beginning. A possible hierarchy is shown in figure 1 (Only some nodes are named). We can see how the hierarchy facilitates abstraction and task control. Also, the nature of most design projects is such that changes are often made. That is where a dynamic hierarchy becomes essential. It might be decided that it would be better to make the 'drive' transaction a transaction group and split it into two. We would then need the facility to abort the drive transaction and create a transaction group with two children instead.

3.2 Multi-copy versus Single-copy system

Regardless of the specification of the correctness criteria and the way we implement them, we have to make an important decision regarding the number of copies of an item that are

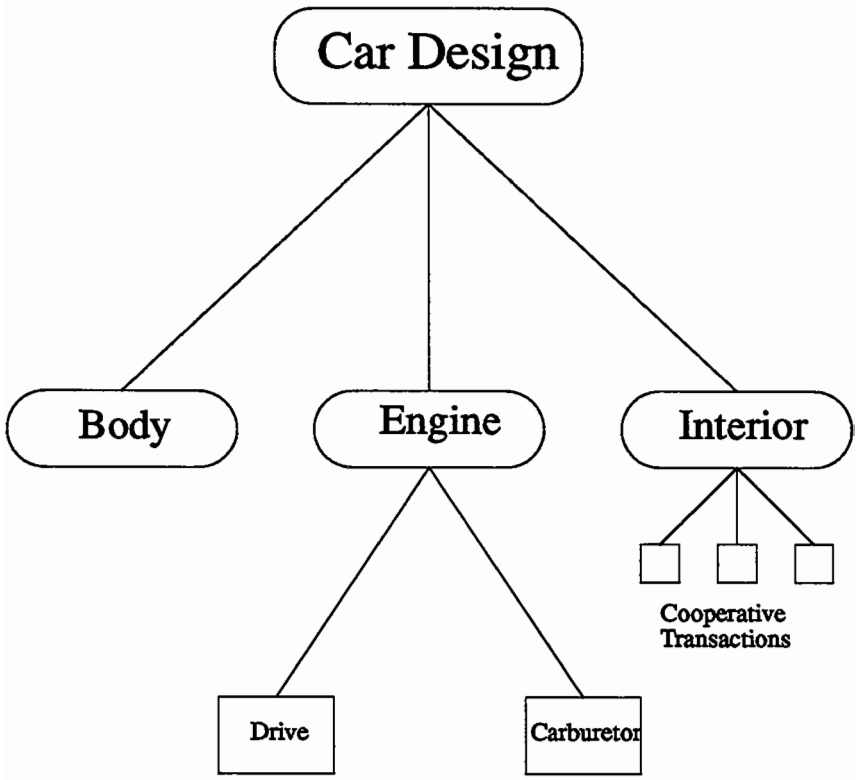

Figure 1: A Hierarchy

maintained in the database. Keeping only one copy of an entity has its advantages. We don't have to keep track of the many copies that would arise in practice if we allowed a copy to be kept in the cache of each and every transaction group. Merging and garbage collection can be completely avoided.

On the other hand, having only one copy has its disadvantages. Since many members may be accessing this copy simultaneously, we have to validate an operation all the way up to the root whenever an operation is issued. This is because the copy will have to kept at the root of the hierarchy where it can be accessed by all the transactions. And therefore, we will be examining the issued operation against the correctness criteria of each ancestor of the member that issued the operation. As we can see, this violates our intuitive notion of abstraction that each subtransaction and transaction group is supposed to bring about. Further, synchronization is beginning to sound complex, because each and every operation needs to be authenticated at a whole path of nodes. And all this work will be wasted if the transaction issuing the operation fails. We feel that it is well worth the extra effort to maintain extra copies of a data item in a cache in a TG.

To elaborate, all items accessed by a CT have copies made available to them in the cache of their parent. Thus the TG projects a complete environment, a virtual database, to each of its children. These copies will in turn be presented to the TG's parent when the TG wants to commit. At this point, they will be merged with the copies that the TG's parent maintains. When a top-level transaction commits, its changes are made by the root transaction to the database and are persistent.

3.3 Correctness Criteria

In the domain of cooperative applications, there is no single criterion like global consistency and atomicity that supports cooperation. Hence, we have to allow user-specified correctness criteria.

A TG's correctness criteria describe its valid histories. A history is a sequence of operation invocations issued by its member CT's. A TG's history in turn forms a part of its parent's history.

In [Skar89] a notion of *patterns* and *conflicts* is proposed. Patterns indicate sequences of operations that are allowed (sometimes necessary) in the history of the member. Conflicts indicate sequences that are forbidden in its history. These specificational criteria are localized to members of a single group. Interaction between two arbitrary points in the transaction hierarchy is controlled by the specifications at their least common ancestor. We now call a history of a CT *correct* if it satisfies all its patterns and conflicts. A history of a group is correct if it satisfies the criteria for the group and the subhistories are respectively correct for each member of the group.

We now see that concurrency control among members of the transaction hierarchy boils down to enforcing the user-specified correctness criteria. Here we face an inherent trade-off. If the specification mechanism is very restrictive, say we use only regular expressions, enforcing these can be done efficiently. On the other hand, if we allow arbitrary programs to specify the interleaving, enforcing the specifications can turn out to be very hard. "Good" correctness criteria are those that allow the user flexibility in specifying how transactions behave and interact, while being efficiently implementable in practice.

Enforcing a specification can be done in two ways. In the optimistic way, we wait for the transaction to issue its whole set of operations first. We then check this sequence against its specifications. This could lead to a lot of wasted work in an environment where iteration and mistakes are inherently part of the development process. It would be much better if we could enforce the criteria *online*. That is, we inform the user immediately if s/he submits an invalid operation. This means that the algorithm that we use to enforce the criteria should not only be able to recognize valid sequences of correct histories but should also be able to able to recognize *valid prefixes* of correct histories. To a great extent, our ability to do this is dominated by the freedom allowed in the specification mechanism. We will say that a specification mechanism has the *viable prefix property* if we can recognize all valid prefixes of its members online.

Further, given a specification, we would like to find out whether it is *consistent* (i.e., it accepts a non-empty history). Here, we are speaking only of local consistency. That is, do the specifications for a transaction include a non-empty set of histories? The problem of global consistency, that of a whole hierarchy accepting a non-empty history, is much harder. It depends not only on the individual nodes, but also on how they interact.

We still have to have a mechanism for this specification. We examine some methods that have been suggested. And then we propose a more powerful and flexible specification mechanism.

4 Specification of Patterns and Conflicts

In order to implement the user-specified criteria, it is clear that we have to make sure that a history is a member of all the patterns that have been specified and a non-member of the conflicts. In other words, it is a very basic necessity that we are able to determine membership in a pattern and membership in the complement of a conflict efficiently.

[Skar89] proposes the use of *pattern machines* to specify patterns and conflicts. A pattern machine is a finite state automaton (AFSA) whose power is augmented with the addition of local variables, the ability to evaluate predicates and the ability to perform actions. A transition occurs from one state of the AFSA to another if the current state has an outgoing arc labeled with the current operation. (If it does not, then that AFSA does not do anything. Thus patterns and conflicts are specified over projections of correct histories rather than exact histories themselves.) Also, the FSA can perform an arbitrary action on this transition.

In this case, determining membership in a pattern is r.e. hard and determining non-membership in a conflict specification is co-r.e. hard, both of which are very hard problems. It is also not possible, in general, to recognize viable prefixes because of the power of the mechanism.

[NFSZ90] proposes the use of a more restricted mechanism for concurrency control. Here pattern machines are specified by deterministic finite state automata (DFSA) that are not allowed to have any local variables or take arbitrary actions on transition. Even here, patterns and conflicts are defined over projections of correct histories rather than exact histories. That is, if at a state, there is no outgoing arc labeled with the current operation, we do not perform any action. The top half of figure 2 shows a machine that enforces strict two-phase locking with respect to an object.

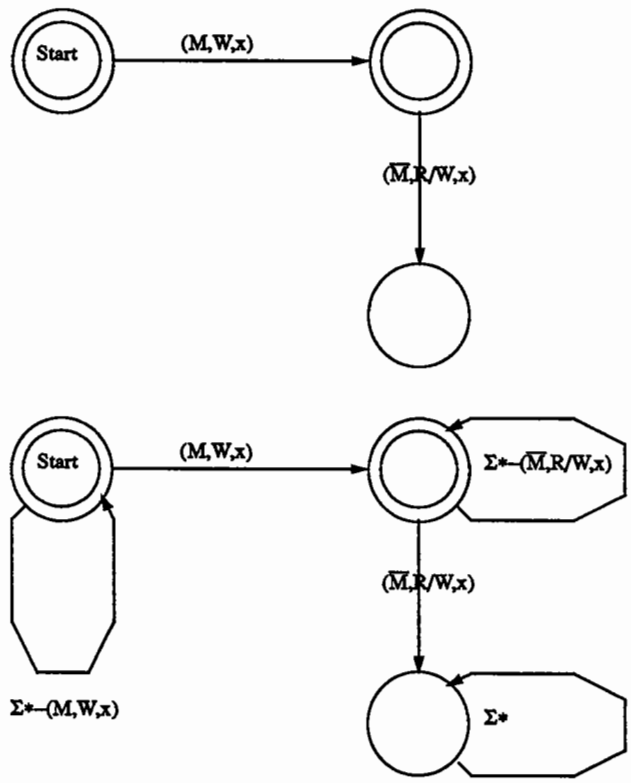

Figure 2: Augmenting DFSA's

We would like to examine the computational complexity, viable prefix property, etc. of this mechanism. First, we augment the machines to make them accept complete sequences of histories rather than projections of correct histories. We accomplish this by adding reflexive arcs in each state for every operation that has no outgoing arc. Now, all the pattern machines accept complete strings over the same alphabet. The lower part of figure 2 shows the locking machine augmented to accept complete histories.

The first observation that we make is that patterns and conflicts, as specified here, are closed under intersection and complement. This means that all the conflict machines can be converted into pattern machines.

Now, it is easy to see that recognition of a history of length n with m pattern machines can be accomplished in $O(mn)$ time. That is, we can recognize histories efficiently. Also, it is easy to check viable prefixes, because a transition to a state from where the final state is not reachable is easy to determine. Therefore, this mechanism has the viable prefix property and we can recognize histories online. Unfortunately, checking consistency, even in this very restricted case is PSPACE-complete. (This problem is that of determining non-emptiness of the intersection of deterministic finite state automata and is PSPACE-complete [GaJo78]).

4.1 Patterns and Conflicts with Grammars

We now give a simple example of a case where the power of the restricted DFSA's is not enough. Let us suppose that a transaction is in charge of opening and closing files. It issues a *inc(o)* on a numerical object *o* whenever it opens a particular file and issues a *dec(o)* when it closes a file (inc and dec stand for increment and decrement respectively). The consistency constraint that we want to express is that the number of closes should be the same as the number of opens issued and the transaction cannot commit until all opens have been closed. This constraint cannot be expressed (for arbitrary number of opens and closes) with DFSA's. On the other hand, we can write a Context Free Grammar that satisfies this constraint easily.

$$\mathcal{S} \longrightarrow \mathcal{A}c$$
$$\mathcal{A} \longrightarrow \mathcal{A}\mathcal{B} \mid \mathcal{A}$$
$$\mathcal{B} \longrightarrow inc(o)\mathcal{A}dec(o) \mid inc(o)dec(o)$$

$\mathcal{S}$ is the start state and c is the commit operation. A string is derivable from this grammar only if it contains "balanced" parentheses.

But even with this more powerful mechanism, we would like to preserve the viable prefix property. We chose the class of grammars called LR(0) grammars. ([HoUl79]. LR(0) stands for "left-to-right scan of the input producing a rightmost derivation and using 0 symbols of lookahead on the input.") Deterministic Context Free Languages can be written in this form. We choose to have 0 symbols of lookahead because, we cannot look ahead in a transaction environment.

This kind of specification has some useful properties. It has the viable prefix property. Further, DCFL's are closed under complementation. This means that we can once again

simplify our problem from having patterns and conflicts to having patterns only. Since we can determine membership in a single grammar efficiently ([HoU179]), we can determine membership online in a set of these grammars.

The price to be paid for this augmented power is that consistency checking, which was PSPACE-complete before, becomes *undecidable* in the general case. (The problem is that of determining non-emptiness of the intersection of DCFL's. It is shown to be undecidable even with two DCFL's in [HoU179].)

4.2 Recovery and Criteria Specification

Now we examine the impact of the power of patterns and conflicts on recovery. As we remarked before, design transactions tend to be very long and it is too expensive to treat them as atomic. What the recovery mechanism should do is undo as few operations as possible so that all the patterns (in our scheme and in the scheme with FSA's, conflicts can be converted into patterns because of closure under complementation) are left in a "correct" state.

Before we define what we mean by a "correct" state, we give a definition for what we call an "invalid" operation. An operation is invalid if:

1. It is part of a transaction that failed
2. It "reads from" an operation that failed. An operation is said to read from another operation when it reads the value written by the former.
3. It is in "front of" an operation that failed. An operation is said to be in front of another operation when either of the following happen. First, the operation is ahead chronologically of a failed operation in a transaction. Second, the operation is ahead chronologically of a failed operation in a pattern.

We can combine these relations ("reads-from," and "front of") to obtain a single "follows" relation. Now, we can say that the recovery process should leave all the pattern machines in a state such that:

1. A final state is reachable.
2. There are no invalid operations that have been accepted in any pattern.

With this definition, we can say that recovery consists of examining the follows relation and undoing the effects of invalid operations. This can be accomplished in the case of FSA's using a log ([NFSZ90]) that stores these relationships and transitions in persistent storage. We can handle recovery when we use grammars to specify correctness similarly. The added complication with grammars is that in addition to storing the relationships and state transitions, we also have to store the stack that is necessary to recognize grammars.

4.3 Partial Undo's

We now introduce an important concept for making design transactions more flexible. Often, a designer makes some mistakes and would unlike to undo some of the operations that s/he submitted. The problem with handling this undo is that other transactions might have submitted operations based on these operations. They will also have to be undone and the owners of those transactions informed.

Undoing a set of operations is very similar to recovering from an abort or a system crash. We handle undo by a process that is essentially similar to that of the recovery process. We invalidate a set of operations based on the set that is submitted to be undone and inform all the transactions that are affected by this undoing process. We believe that this gives an added degree of flexibility to the design process.

5 A Transaction Control Model

We now give a precise definition of the transaction model and discuss some implementation details.

5.1 The Hierarchy and Correctness Criteria

Transactions are organized in the form of an hierarchy that is dynamic. The root transaction is run by the system and is in charge of maintaining the database. Other transactions run under the root at some point in the hierarchy. The root separates the top-level transactions (which are unrelated to each other) usually by enforcing strict two-phase locking.

There are two kinds of transactions, namely, Transaction Groups (TG's) and Cooperative Transactions (CT's). A Transaction Group is a collection of Transactions and contains specifications for the correctness of its members. It is created when there is a request placed at the database server. This request specifies the point in the hierarchy where the TG is to be placed and also gives the correctness specifications of its members. In addition to the correctness specifications, it might also contain specifications for abstraction. We will come back to these shortly.

A CT is the basic unit of a transaction in the database. Unlike in traditional applications, it is not expected to be atomic or correct in a global sense. The parent of the CT is in charge of ensuring that the history produced by its children are correct. This parent also maintains a cache of the items accessed by its children and appears as the complete database to its children.

5.2 Correctness Criteria

Correctness Criteria are specified for a TG when it is created. These can be either be in the form of grammars or in the form of finite state machines. The system then converts these into an internal format suitable for enforcing concurrency control. In an implementation, we expect most of the commonly used criteria to be available in the form of libraries. Using the ready-made templates, the creator of the TG fills up the necessary specifications which are then converted to the actual specifications using a pre-processor. This pre-processor might also detect obvious inconsistencies and report them to the user.

5.3 CT's and Object Bases

Since we expect design transactions to run on object bases rather than a traditional database, we would expect a CT to issue operations that are methods on the underlying object base. Unfortunately, they are some fundamental problems that we have not yet answered. (For example, in a general database, it is very difficult to write a program that will answer the question, "Are the following two sequences of methods equivalent?" in the sense that they have the same effect on the database. Such a question is easily answered with just reads and writes [Papa79].) We are currently working on a model that allows us to answer questions like this efficiently. For now, we restrict the operations that can be issued by a CT to the set $\{read, write, inc, dec\}$.

Further, we would like the natural concept of abstraction available in object bases to be present in the transaction control system. That is, a set of reads and writes should be replaced by the equivalent method on the underlying object base. And a set of methods applications by a child appears in its abstracted form at its parent. Currently, we restrict ourselves to simply condensing a bunch of writes into a single equivalent write and a bunch of reads into a single read.

5.4 Commit, Abort and Recovery

When a transaction wants to commit, it places a request with its parent. The parent verifies that the child satisfies all the correctness criteria and then allows it to commit. The changes made by the child are now permanent with respect to its parent's cache. If the parent aborts, the child's work will disappear. Therefore, it is clear that changes are permanent at the database only when a top-level transaction commits.

A child can abort voluntarily without its parent's permission. After the abort, the parent invalidates the operations submitted by aborted child and then using the log invalidates the operations that "follow" these operations and informs the members that might have been affected. Thus a TG is in charge of coordinating recovery among its members.

5.5 Versions, Locking

Since we maintain a cache at each TG, there can be many versions of the same object at the same time. Version management comes naturally because of the hierarchy. When a TG commits by submitting its versions to its parent, then we can remove of all its versions. Changes are made to the database copy of the object when a top-level transaction commits.

Even in a cooperative environment, we would like to allow for locking because a transaction might want to ensure that it can make changes to a particular object. We allow it to do so by making it request locks at startup. The parent checks its patterns and makes sure that the child can access these objects and grants the permission to the child to make changes to these objects. The child can make these changes knowing that they will be permanent with respect to its parent.

6 Summary and Extensions

We have presented an analysis of transaction control systems for design environments. We have examined the impact of relaxed constraints on the computational complexity of concurrency control and have argued that consistency checking, even for simple cases is very hard. We have presented a method of relaxing constraints that is very powerful and subsumes most current models, while preserving the ability to do concurrency control and recovery efficiently.

We plan extend this work by placing the transaction control system on top of an object base. We also plan to examine more sophisticated constraint specification mechanisms that can have efficient implementations.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[HoUl79] J. E. Hopcroft, J. D. Ullman. *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[GaSa87] H. Garcia-Molina, K. Salem. "Sagas," In *ACM SIGMOD Proceedings*, ACM, 1987.

[GaJo78] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[KKB85] H. F. Korth, W. Kim, F. Bancilhon. "On long-duration CAD transactions," *Information Systems*, 13, 1987.

[KoSp88] H. F. Korth, G. D. Speegle. "Formal Model of Correctness without Serializability," In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1988.

[KLS90] H. F. Korth, E. Levy, A. Silberschatz. "A Formal Approach to Recovery by Compensating Transactions," In *Proceedings of the 16th VLDB Conference*, 1990.

[Lync83] N. A. Lynch. "Multilevel Atomicity — A new Correctness Criterion for Database Concurrency Control," ACM *Transactions on Database Systems*, December, 1983.

[Moss85] J. E. B. Moss. *Nested Transactions: an Approach to Reliable Distributed Computing*, MIT Press, 1985.

[NFSZ90] M. H. Nodine, M. F. Fernandez, A. Skarra, S. B. Zdonik. "Cooperative Transaction Hierarchies," Technical Report, Brown University, February 1990.

[Papa79] Papadimitriou, C.H. "Serializability of Concurrent Database Updates," *Journal of the ACM*, October 1979.

[Skar89] A. Skarra. "Concurrency Control for Cooperating Transactions in an Object-Oriented Database," *SIGPLAN Notices*, April 1989.