Transformational Concurrency Control
for Collaborative Design and Arrangement

Tom Hudson
First Release: 5 May 1997
Tech Report: 28 May 1999

May 27, 1999

Contents

List of Figures

1 Introduction

Distributed databases traditionally use pessimistic approaches to concurrency. Two sites attempting to change the same piece of data at the same time can produce chaos and inconsistiencies. Thus, each site must lock the data it wants to use. Optimistic concurrency control was introduced by Kung and Robinson[9], and has found increasingly common use in the ensuing 16 years.

Virtual reality and three-dimensional graphics must use some form of database to store the state of the scene they render. They have not yet seen much use of optimism. Commercial distributed VR toolkits, like dVS/dVISE from Divison, typically use centralized databases. The distributed database solutions now emerging from research labs into mass market applications commonly require that all writes to a given object be funneled through a single site, so that they are serialized and controlled. Their replication reduces read latency, but not write latency.

We propose to extend optimistic concurrency control to the databases used for interactive graphics. We begin with an approach from the CSCW community, Transformational Concurrency Control. With this family of algorithms, the state at each site in a distributed system is not necessarily consistent. Every operation received from remote sites is *transformed* into a different operation to be executed locally. The rules governing transformations are constructed so as to ensure that after all operations have been executed at all sites, the state of the database is globally consistent.

Practical Domain: Designers building complex systems such as submarines or satellites are forced by the magnitude of their task to work in parallel, with several users laying out different systems in the same region of the design at the same time. Currently, it is not uncommon for awareness of one another's work to suffer an overnight delay; if several users make incompatible changes to the database at 8am, they will not find out about it until the following day, at which point they will have to meet and renegotiate their use of that space, each compromising as much as an entire day's work.

Ideally, users would be aware of each other's work, so that they could adapt to one another's needs for the space (which they may not, and practically *can not*, be aware of in advance and in detail). However, we do not want to force all conflicts to be resolved immediately. Promptly alerting users of a problem but allowing them to delay resolving it leverages the strengths of the computer: automatic detection of a conflict is fairly easy, but knowing the right way to automatically resolve such a problem is hard.

An important tool for preparing variants of designs and exploring alternatives is versioning. Our algorithms support two forms of versioning. First, explicit versions: When a significant set of changes has been made, a user may choose to checkpoint the database, storing the current state of the design stably. In the future, users may revert to this state for their work, whether to see how the design evolved or to explore the design space in another direction from this past point. Thus, a tree of versions is formed. Second, implicit versions: When multiple users working simultaneously begin a series of actions that are incompatible with one another (in a sense defined by the application), versions are automatically created to hold all of the conflicting modifications.

Main Contribution: We extend transformational concurrency control to a new application domain. This domain, spatial arrangement and design, has a significantly different semantics than the domains in which transformational concurrency control solutions have typically been applied. We show how to use the same framework to support group undo (as did Ressel et al.[16]), and use it for the new purpose of supporting both implicit and explicit versioning.

2 Related Work

There are three broad areas of related work – collaborative systems (both applications and infrastructures), which give us concurrency control and group undo mechanisms, as well as some work in versioning; CAD databases, which set norms for functionality in versioning systems for design objects; and Virtual

Reality, which gives us latency and user interface constraints. It is significant that versioning for design objects is typically far more elaborate than the versioning addressed by collaborative systems.

2.1 Collaborative Systems

2.1.1 Concurrency Control

The two sources in Computer-Supported Cooperative Work on which we draw most directly are the dOPT[6] and adOPTed[16] algorithms. In fact, this paper draws quite heavily on Ressel et al. (adOPTed), and we use their formal treatments and notation throughout, quoting their statement of the algorithm in Figures 3 – 4.

These are not the only works to use optimistic concurrency control to handle coordination of several users. A notable commercial successe is Lotus Chronicle, a distributed, shared spreadsheet. When multiple users try to change the same part of a spreadsheet simultaneously, a version is automatically created for each user. Users can view one another's alternatives and use whatever means they choose to merge or select between them. Lotus Notes uses a similar, coarser-grained policy.

The Coda file system supports networked data stores which are strongly connected, disconnected, or in an intermediate state ("weakly connected") where bandwidth is low and costs are high. Concurrency control is asymmetric: strongly-connected systems update one another continuously. Weakly-connected systems and send batches of updates to propagate across the strongly-connected network, and do not receive any updates from other systems. This optimistic scheme means that conflicts occur and merging is necessary; there is some support in Coda for automatic merging, as well as manual.

Rover is a distributed object system with the same network model as Coda. It supports migration and replication of objects, even in disconnected or weakly-connected operations, by an asynchronous RPC scheme. Since multiple replicas of an object will exist (unless locking protocols are used), Rover also needs merging support.

2.1.2 Versioning

Borges and Jomier[5] have a complex, useful system to support group awareness in exactly the manner in which we are interested. To support awareness it is necessary to record not only changes, but also the reasons for the changes and the processes used to effect the changes. Different users want different levels of awareness of different things, and their system attempts to automatically track this. Each user's knowledge and intention is modeled by the system and drives the amount of detail given in the response to their queries.

Haake, Haake, and Hicks[7] have proposed several models of interfaces for manually merging hypertext networks. At the software implementation level, 3D designs are often represented as scene graphs, which we could model similarly to hypertext networks and thus use the same interfaces for. However, this representation is most meaningful to 3D graphics programmers, and does not necessarily match the mental model used by actual designers.

Versioning, as with many other high-level collaboration coordination mechanisms, can be either coarse-grained or fine-grained. Although fine-grained is more flexible, Melly and Hall[14] point out that coarse-grained version control may offer significant ease of use. In addition to being lightweight, coarse-grained versions are very simple, small in number (where fine-grained versioning may create extremely large numbers of versions of each object), and always directly traceable to a single significant event – typically a user-initiated versioning process.

Wieczerzycki[17] gives a complex database semantics in order to support CSCW, hoping to target it at our very problem – Computer-Aided Design. However, much of his paper is spent applying the semantics to modeling workflow and to adapting long-duration transactions from the database literature to a versioning environment. His related work in workspaces, object percolation, and merging should be a significant influence on the realization of efforts like ours.

2.1.3 Undo

Prakash and Knister[15] give an undo mechanism that is very similar to transformational concurrency control; they cite the GROVE editor that implemented dOPT, but do not seem to be aware of the similarities between their own approach and that introduced by Ellis and Gibbs.

2.2 CAD Databases

Since traditional database systems are "designed to deal with only regular and structured data with no built–in concepts of versioning"[1], numerous papers have been written to handle the problem of storing versioned, hierarchical objects in CAD. Most of these, however, deal with database infrastructure, rather than how users are to create, view, and manipulate versions. A good survey of this work is in Katz[8].

The issues discussed for database versions are relevant to our explicit versioning process. However, they all occur at a lower level than this paper addresses. The underlying version system's capabilities do not directly affect the algorithms discussed in this paper, but rather affect the work practices of the users.

Much of the complexity of Wieczerzycki's paper mentioned in the previous system would seem to stem from the problem of storing versioning data in a traditional database system, which Ahmed and Navathe point out is difficult[1].

2.3 Virtual Reality

CALVIN[11, 12] is a system supporting transcontinental collaboration in the architectural design of spaces. It maintains the state of a design in a central database, acknowledging that this limits the number of remote clients by "bandwidth and latency". It uses ATM to partially offset this limitation, and accepts the remainder since it is a prototype meant to deal with interaction rather than distributed issues.

A system descended from CALVIN has been used for collaboration on the design of farm vehicles between Americans and Europeans. Originally it had no concurrency control at all, which proved problematic; at a later date locking was added[13].

CALVIN and related systems maintain a linear set of versions: every time the database is saved, it does not overwrite the previously saved record. Users can revert to earlier copies of the database, but new additions are appended to the end of the history, rather than on a side-branch.

dVS/dVISE are commercial products of Division, Inc. They support distributed virtual reality with a centralized database. Since all writes must go through this database, concurrent operations need not be explicitly coordinated; serialization is automatic.

Universal Worlds is a proposal whose authors include the researchers at Mitsubishi Electronic Research Labs who worked on SPLINE (See e.g. [2, 3]). It has emerged as one of the leading contenders for a standard infrastructure supporting VR over the Internet. As do many of its proprietary peers, it uses a fully replicated database at each site involved in a VR session. However, writes to each *object* in the database are centralized – only one instance of the object is writeable. Write requests are sent to the host currently holding the writeable object, or a distributed protocol is used to migrate the capability to write to the host desiring to make a modification.

3 The Basic Model

3.1 Semantics and Operations

Our model of an assembly or arrangement application has two components:

- **Objects**: A set of objects O, P, ...
- **Space**: A discrete space with positions X, Y, ...

On this model we define two operations:

- **Insert**: $Ins(O, X)$
  Insert object O into the space at position X.

- **Delete**: $Del(O, X)$
  Delete object O from the space at position X.

A user performs an operation by issuing a *request*; requests are the units of network transmission. A request is a 4-tuple containing:

- u The id of the user issuing the request
- k The serial number of the request: the total (inclusive) number of requests issued by that user
- v The state vector specifying the system seen by the issuer. (State vectors are defined in section 5)
- o The operation to be performed.

3.2 Constraints

Our intent is to address arrangements or design for the real world. This gives us one significant constraint that will shape our entire approach: no two objects may overlap.

We are actually addressing the problem of placing objects into a continuous 3-space. We assume that there is a function similar to Prakash and Knister's $Conflicts(X, Y)$[15] that reports whether or not two volumes overlap – a collision-detection function. Using this function, and only worrying about pair-wise interactions between objects, we reduce the complexity of space we need to treat from three continuous dimensions to one discrete dimension.

3.3 Implicit Versioning

Since we constrain the state of the application, some operations (e.g. an insert of an object that would overlap another already present) may be illegal. However, we are using optimistic concurrency control. This means that different instances of the application may have different views of the state. Users can issue operations that, when the application quiesces and converges, are found to conflict. Since the users issued the operations "simultaneously", neither has priority, neither should be undone automatically by the system. Instead, we need to automatically, or *implicitly*, create one version for each user. The implicit versions show the differences between the users' requests. However, each version needs to show all operations by either user that do not conflict.

To capture the fact that an insert was requested that conflicts, we extend our set of operations to include "Conflicted Insert". This new operation explicitly contains the id of the user who made the request and the set of other operations it conflicts with.

- **Conflicted Insert**: $Ins_A(O, X, \{G\})$
  User A attempts to insert object O into the space at position X, in conflict with the set of operations G.

Using implicit versioning like this allows us to make the common case (non-conflicting operations) fast, and use a (computer-supported) social protocol to reconcile when optimism fails. Further, extending our set of operations, our model, to include versioning captures all the implicit versions and saves them with the database. This permits users to defer resolution of conflicts to a more convenient time, perhaps when the remainder of the design is more clear, or when others can be brought in to arbitrate the conflict.

3.4 Transformation Rules

Every conflict involves two operations. There are two possible approaches to writing transformation rules for them, which we refer to as Ignoring Equality and Rewriting History.

If the set of possible operations is R, we define a transformation function $tf()$ with signature $R \times R \rightarrow R \times R$[1]. We also define tf_1: $tf(r_1, r_2) = (r_1', r_2') \Rightarrow tf_1(r_1, r_2) = r_1'$. This is the operation into which r_1 is translated if it occurred concurrently with r_2 and is locally being serialized *after* r_2.

3.4.1 Solution: Ignoring Equality

In the adOPTed algorithm, once an operation is executed at a site it is fixed, unmodifiable. If we choose this, only one of each pair of conflicting operations can be recorded as a Conflicted Insert, since the other will have already been executed as an Insert.

This requires that we relax the definition of equivalence used in the convergence property. The two sequences given in Figure 1 must result in equivalent states; that is, we need to define equivalence in our system such that the two very different states resulting from the two sequences of instructions in Fiture 1 are equivalent. Both states contain the same basic operations: two inserts, of O and P, both at X. They differ in order (and thus in which is labeled as conflicted).

If we follow the adOPTed algorithm[16], relaxing equivalence, we arrive at the following set of transformation rules:

$tf_1(Ins(P, X), Ins(O, X)) = Ins_B(P, X, \{Ins(O, X)\})$
$tf_1(Ins(O, X), Ins(O, X)) = Shadow(Ins(O, X), \{Ins(O, X)\})$
$tf_1(Del(O, X), Del(O, X)) = Shadow(Del(O, X), \{Del(O, X)\})$

$Shadow(r, G)$ is defined similarly to Ressel et al.: the request r cannot be executed so long as the set of conflicting requests, G, is nonempty. If a request

[1]Formally, tf is a function from requests to requests, but throughout we assume the first three values of the request are known and only deal with the operation.

in G is undone, it is removed from G. (If G is then empty, r is ready to be executed). Any other request that would conflict with r may be added to G.

For the JointEMACS[16] text editor, it was necessary to add every subsequent operation to G if it was not an undo of an operation already present in G. To increase the effectiveness of the Undo command (since we are not using a linear Undo), we avoid this conservatism and add only the necessary operations to the conflict set G.

3.4.2 Solution: Rewriting History

The above solution requires redefining equivalence between states.

Instead, we may want to rewrite history: change the log of executed events. This formally may require undoing and redoing long chains of events, although the practical implementation is far more straightforward.

Instead of using $tf()$, since we are rewriting history and changing both arguments, we define a similar function $xf()$.

The first of the transformation rules is the only one that must change, but since we are using a different notation here we repeat all three:

$$\begin{aligned}
xf(Ins(O,X), Ins(P,X)) = \; & (Ins_A(O,X,\{Ins(P,X)\}), \\
& Ins_B(P,X,\{Ins(O,X)\})) \\
xf(Ins(O,X), Ins(O,X)) = \; & (Ins(O,X), \\
& Shadow(Ins(O,X),\{Ins(O,X)\})) \\
xf(Del(O,X), Del(O,X)) = \; & (Del(O,X), \\
& Shadow(Del(O,X),\{Del(O,X)\}))
\end{aligned}$$

4 Motion

Editing an arrangement with only Insert and Delete operations is awkward and unnatural. Instead, what the users we have observed typically do is insert an object and then adjust it: move it slightly, rotate it somewhat, or resituate it completely. To facilitate this, it seems natural to extend the set of operations to include a Move command.

$Ins(O,X)$
$Ins_B(P,X,\{Ins(O,X)\})$

$Ins(P,X)$
$Ins_A(O,X,\{Ins(P,X)\})$

Figure 1: Two sequences of operations that must result in equivalent states.

$Ins_A(O, X, \{Ins(P, X)\})$
$Ins_B(P, X, \{Ins(O, X)\})$

$Ins_B(P, X, \{Ins(O, X)\})$
$Ins_A(O, X, \{Ins(P, X)\})$

Figure 2: Two sequences of operations that are equivalent, but require rewriting history.

- **Move**: $Mov(O, X, Y)$
  Move object O from position X to position Y.

4.1 Unbundling Mov

A move can be thought of as an atomic delete - insert pair: $Mov(O, X, Y)$ is equivalent to $Del(O, X); Ins(O, Y)$. This would add to the simplicity and elegance of our system, since the transformation table only needs to produce Delete and Insert; we never need to have conflicted move operations, although we will need to introduce conflicted delete.

- **Conflicted Delete**: $Del_A(O, X, \{G\})$
  User A attempted to delete object O from the space at position X, conflicting with the set of operations G.

Previously, we've been able to handle every case of multiple deletes of the same thing by using $Shadow()$. However, $Shadow()$ completely supresses the effects of first argument, while we want to show that the Delete operation in question has been executed in at least one implicit version.

Although we break the *execution* of a Move into two operations, while processing it through the transformation table we keep it unitary. This is because decomposing it loses information. In fact, we will see that this information loss can bring us to grief and force adoption of a more complex set of transformations.

4.1.1 Ignoring Equality

If we ignore equality, we obtain the following transformation rules;

$$tf_1(Mov(O, X, Z), Mov(O, X, Y)) = \begin{array}{l} Shadow(Del(O, X), \{Del(O, X)\}); \\ Ins_B(O, Z, \{Ins(O, Y)\}) \end{array}$$

Since both users desire to move the object, using $Shadow()$ here rather than a conflicted delete matches the behavior expected by users. The difference in their intentions is where to move O, not whether to move it.

$$tf_1(Ins(P,Y), Mov(O,X,Y)) = \quad Ins_B(P,Y,\{Ins(O,Y)\})$$

$$tf_1(Del(O,X), Mov(O,X,Y)) = \quad Del_B((O,X,\{Del(O,X)\})$$

In this case, $Shadow()$ is inappropriate, since one user intends to delete O and the other wants to move it.

$$\begin{aligned} tf_1(Mov(O,X,Y), Ins(P,Y)) = \quad & Del_B(O,X,\{Ins(P,Y)\}); \\ & Ins_B(O,Y,\{Ins(P,Y)\}) \end{aligned}$$

$$\begin{aligned} tf_1(Mov(O,X,Y), Del(O,X)) = \quad & Del_B(O,X,\{Del(O,X)\}); \\ & Ins_B(O,Y,\{Del(O,X)\}) \end{aligned}$$

4.1.2 Rewriting History

If, instead, we rewrite history, we have the following transformations:

$$\begin{aligned} xf(Mov(O,X,Z), Mov(O,X,Y)) = \quad & (Shadow(Del(O,X),\{Del(O,X)\}); \\ & Ins_B(O,Z,\{Ins(O,Y)\}), \\ & Del(O,X); \\ & Ins_A(O,Y,\{Ins(O,Z)\})) \end{aligned}$$

$$\begin{aligned} xf(Ins(P,Y), Mov(O,X,Y)) = \quad & (Ins_B(P,Y,\{Ins(O,Y)\}), \\ & Del_A(O,X,\{Ins(P,Y)\}); \\ & Ins_A(O,Y,\{Ins(P,Y)\})) \end{aligned}$$

$$\begin{aligned} xf(Del(O,X), Mov(O,X,Y)) = \quad & (Del_B(O,X,\{Del(O,X)\}), \\ & Del_A(O,X,\{Del(O,X)\}); \\ & Ins_A(O,Y,\{Del(O,X)\})) \end{aligned}$$

This is another strong example of why we can't simply $Shadow()$ the Delete, but must instead have a Conflicted Delete operation.

$$\begin{aligned} xf(Mov(O,X,Y), Ins(P,Y)) = \quad & (Del_B(O,X,\{Ins(P,Y)\}), \\ & Ins_B(O,Y,\{Ins(P,Y)\}); \\ & Ins_A(P,Y,\{Ins(O,Y)\})) \end{aligned}$$

$$\begin{aligned} xf(Mov(O,X,Y), Del(O,X)) = \quad & (Del_B(O,X,\{Del(O,X)\}); \\ & Ins_B(O,Y,\{Del(O,X)\}), \\ & Del_A(O,X,\{Del(O,X)\})) \end{aligned}$$

4.2 Why We Can't

Rewriting a Move operation as a Delete followed by an Insert is handy, since it reduces the number of operations to be implemented to two, but it has a number of faults.

- It may break the NoOp property. (See Section 5.2)
- It breaks user expectations when we allow undo.
- It suffers from ambiguity: If user A executes $Mov(O, X, Y)$ and user B executes $Del(O, X); Ins(O, Z)$, we have a different set of transformations than if B had executed $Mov(O, X, Z)$.

Most of these result from the information lost when we do the rewriting: information about intent, and information about the relationship between the two operations. So, instead of rewriting Move, we need to keep it in our toolbox as a primitive operation. This means that we will also need a conflicted move:

- **Conflicted Move**: $Mov_A(O, X, Y, \{G\})$
  User A attempted to move object O from position X to position Y, conflicting with the set of opreations G.

This yields the following two sets of transformations:

4.2.1 Ignoring Equality

$$tf_1(Mov(O, X, Z), Mov(O, X, Y)) = \quad Mov_B(O, X, Z, \{Mov(O, X, Y)\})$$

$$tf_1(Ins(P, Y), Mov(O, X, Y)) = \quad Ins_B(P, Y, \{Mov(O, X, Y)\})$$

$$tf_1(Del(O, X), Mov(O, X, Y)) = \quad Del_B((O, X, \{Mov(O, X, Y)\})$$

$$tf_1(Mov(O, X, Y), Ins(P, Y)) = \quad Mov_B(O, X, Y, \{Ins(P, Y)\})$$

$$tf_1(Mov(O, X, Y), Del(O, X)) = \quad Mov_B(O, X, Y, \{Del(O, X)\})$$

4.2.2 Rewriting History

$$\begin{aligned} xf(Mov(O, X, Z), Mov(O, X, Y)) = \quad & (Mov_B(O, X, Z, \{Mov(O, X, Y)\}), \\ & Mov_A(O, X, Y, \{Mov(O, X, Z)\})) \end{aligned}$$

$$xf(Ins(P,Y), Mov(O,X,Y)) = (Ins_B(P,Y,\{Mov(O,X,Y)\}), Mov_A(O,X,Y,\{Ins(P,Y)\}))$$

$$xf(Del(O,X), Mov(O,X,Y)) = (Del_B(O,X,\{Mov(O,X,Y)\}), Mov_A(O,X,Y,\{Ins(P,Y)\}))$$

$$xf(Mov(O,X,Y), Ins(P,Y)) = (Mov_B(O,X,Y,\{Ins(P,Y)\}), Ins_A(P,Y,\{Mov(O,X,Y)\}))$$

$$xf(Mov(O,X,Y), Del(O,X)) = (Mov_B(O,X,Y,\{Del(O,X)\}), Del_A(O,X,\{Mov(O,X,Y)\}))$$

4.3 Inheriting Implicit Versioning

Consider the operation $Mov(O,X,Y)$. If the local state does not contain O at X – if no execution of $Ins(O,X)$ or $Mov(O,*,X)$ has occurred – this is clearly illegal. What if some remote user has requested an operation to place O at X, only to find a conflict? If $Ins_B(O,X,*)$ is in the local history, the local user has some awareness of there being O at X (Likewise for $Mov_B(O,*,X,*)$). It is reasonable to allow the user to move or delete this object from where it was placed by some other user. Thus, we redefine the meaning of conflicted move and conflicted delete:

- **Conflicted Delete**: $Del_A(O,X,\{G\})$
  Some user deleted object O from the space at position X; it has most recently been placed in conflict as a result of actions of user A; its total conflicted set of operations is G.
- **Conflicted Move**: $Mov_A(O,X,Y,\{G\})$
  Some user moved object O from position X to position Y; it has most recently been placed in conflict as a result of actions of user A; its total conflicted set of operations is G.

Now, any operation performed on an object last modified by a conflicted insert or conflicted move is also conflicted, is tagged with the identification of the user who first initiated the conflict, and has the same set of conflicting actions as that initiating operation.

5 Algorithm and Formal Properties

5.1 Requests

As stated in section 3, a request is a 4-tuple:

$r = (u, k, v, o)$

- u: the id of the user at a site
- k: the number of requests generated at a site
- v: state vector
- o: an operation
- U: set of users
- i, j: variables over U
- n: number of users ($||U||$)
- s: a site's view of the application state
- Q: the queue of pending (generated or received but not yet executed) requests at a site
- L: log of all requests generated or received at a site
- G: n-dimensional interaction model

5.1.1 State Vectors

Every request broadcast to remote sites is accompanied by its state vector v. If there are n users (sites), v is an n-component vector. The ith component of v is the number of requests from site i which had been executed previously. This specifies the state of the site when it was executed. v is a vector clock, as introduced by Lamport[10].

5.1.2 Interaction Model

The interaction model used in execution of the algorithm is the total set of executed and transformed requests seen by a site. Its formal definition is considerably more complex: an n-dimensional graph whose vertices are application states and whose edges are the requests used in execution. A detailed explanation of the interaction model and how it is built is given in the adOPTed paper[16].

5.2 Uniform Properties

Some of the simple formal properties follow regardless of the way we choose to define our transformations.

NoOp: Let r_I be a noop request. For all requests r, $tf(r_I, r) = (r_I, r)$.

- $Mov(O, X, X) = I$

Undo: Let $\overline{r_1}$ be an undo request for r_1. For all requests r_2, if $tf(r_1, r_2) = (r_1', r_2')$ and $tf(\overline{r_1}, r_2') = (\overline{r_1}', r_2'')$, then $r_2 = r_2''$ and $\overline{r_1}' = \overline{r_1'}$.

- $\overline{Ins(O, X)} = Del(O, X)$
- $\overline{Del(O, X)} = Ins(O, X)$
- $\overline{Mov(O, X, Y)} = Mov(O, Y, X)$

5.3 Ignoring Equality

Under the Ignoring Equality assumption, the algorithm used at each site for transformation and execution of requests is the same as that used in adOPTed[16]; we repeat it in Figures 3 - 4.

Equivalence: Two valid sequences p_1, p_2 are equivalent if they produce equivalent states modulo versioning when executed in the same initial state.

Transformation Property I:
If $tf(r_1, r_2) = (r_1', r_2')$, then $r_1 r_2' \equiv r_2 r_1'$. (as in adOPTed)

Symmetry Property:
If $tf(r_1, r_2) = (r_1', r_2')$, then $tf(r_2, r_1) \equiv (r_2', r_1')$.

In adOPTed[16], this last consequent is an equality, not an equivalence. This difference results from the fact that, with the our algorithm, states at the distributed sites are no longer equal but are rather equivalent modulo versioning.

5.4 Rewriting History

If we are to be Rewriting History, the algorithm changes only slightly, but its formal properties change significantly. The only algorithmic change is in the definition and use of $TranslateRequest$, since we need to apply r_i'' as a side effect. See Figure 5.

Equivalence: As in adOPTed, we define two valid sequences p_1, p_2 are equivalent if they produce equal states when executed in the same initial state.

Symmetrical Transformation Property:
If $xf(r_1, r_2) = (r_1', r_2')$ and $xf(r_2, r_1) = (r_2'', r_1'')$, then $r_2' r_1' = r_1'' r_2''$.

This captures portions of both Transformation Property I and the Symmetry Property from adOPTed, without quite matching either.

At this time, our investigation of this model has not yielded an equivalent of Ressel et al.'s Transformation Property II for either Ignoring Equality or Rewriting History[16].

6 Explicit Versioning

Users would like to be able to capture a version, a snapshot on stable storage of the current state of the database, at any point in time and in arbitrary database states. The use of optimistic concurrency control makes this problematic.

One of the principal effects of our concurrency control algorithm is that database state is not globally consistent at any given point in time. State is only equivalent at all sites (for whichever definition of equivalence we are using) after the system has quiesced and converged. We have to incorporate versioning into our concurrency control, just as we do any operation that modifies the state of the database.

One simple method to make sure that the state is consistent in a version is to force a quiescence. Once a site receives a request for a versioning operation, accept no more requests *from that site's user* until the versioning has completed. Only operations that were in transit on the network need to be executed. Once the system quiesces, it is guaranteed to converge on a consistent state, and we can take the version globally.

The drawback with this method is that the version that is agreed upon will not (necessarily) be that which the requesting user saw when the versioning operation was requested. Any concurrent operations will still arrive and be executed before the network quiesces. Thus, if a user attempts to save a version containing certain features, and another user concurrently deletes those features, they will not be contained in the version.

Perhaps the most user–friendly approach is to have an operation named Version. Like any operation, when a user requests it it is executed immediately at the local site, while being broadcast to remote sites for inclusion in their request queues. Immediate execution means that what is saved in a version is exactly what the user expects to have saved in a version. Meanwhile, the fact that v (the state vector) is sent with every request allows us to have consistent remote versions. Every remote site that wants to store a copy of the version can (automatically, outside the user's perception) undo until it reaches the state vector in which the version was generated and save that state.

7 Conclusion

Awareness of the work of other users could possibly effect a significant change on the way complex assemblies are designed. It is certainly a necessary part of the tightly-coupled collaborative design environments that are being built today, such as DARPA's Simulation-Based Design program. Using optimistic concurrency control minimizes the latency in the system, maximizing responsiveness to the designer and thus increasing probability of adoption for the new technology. It also allows conflicts to occur. Conflicts have been found to be *useful* in the current systems, and are currently resolved by social protocols. Since a spatial conflict is a notification to the users of a conflict in their higher-level goals, which are not captured in the system, correctly resolving them requires negotiation in the context of these high-level goals, and thus should be handled manually. (In a Simulation-Based Design environment, the system could check automatically to be sure that all design constraints were satisfied by the manual merge, reducing the fragility of human-driven concurrency.)

To support this tight coupling, we have extended transformational concurrency control into new application domains. We have used it to support undo and two different types of versioning.

In this paper, we have addressed conflicts arising in the database only due to concurrency – failures of optimistic concurrency control. However, this is not sufficiently general. It is necessary to allow willful conflicts. This is mostly a straightforward problem of interface design, with a slight increase in the complexity of the transformation tables. If a user intentionally generates conflicting operations, the conflicted operation can be broadcast over the network just like any other, following the standard algorithm.

7.1 Future Work

Our work on Undo is not included in this manuscript. We could develop a formalism to capture the state of the application, allowing us to rigorously treat equivalence and equality of states. Our work here does not include any counterpart for Ressel et al.'s Transformation Property II. The sets of conflicting operations resulting from inheriting implicit versioning are conservative, and should be able to be liberalized significantly. We have not even given sketches of proofs of correctness for our transformation rules; these could be valuable to those attempting to build on our work. (Our development was hampered by the lack of proofs in some of our predecessors.) With major rewriting, we should be able to restructure the algorithm to avoid side effects in $TranslateRequest()$ while Rewriting History. The probable cost would be a change in the underlying models. Finally, and most importantly, this work is yet to be implemented.

References

[1] R. Ahmed and S. Navathe. Version Management of Composite Objects in CAD Databases. ACM, 1991, pages 218–227.

[2] D. Anderson, J. Barrus, J. Howard, C. Rich, C. Shen, and R. Waters. Building Multi-User Interactive Multimedia Environments at MERL. Mitsubishi Electronic Research Laboratories, Tech Report TR–95–17, November, 1995. (Also appearing in *IEEE MultiMedia*, Winter, 1995.)

[3] J. Barrus, R. Waters, and D. Anderson. Locales and Beacons: Efficient and Precise Support For Large Multi-User Virtual Environments. Mit-

subishi Electronic Research Laboratories, Tech Report TR–95–16, November, 1995.

[4] T. Berlage. A Selective Undo Mechanism for Graphical User Interfaces Based On Command Objects. *ACM Transactions on Computer-Human Interaction*, 1:3, September 1994, pages 260–294.

[5] Borges and Jomier. Using Database Versions to Support Awareness in Group Interactions. In *Proceedings of the ECSCW'95 Workshop on the Role of Version Control in CSCW Applications.*

[6] C. Ellis and S. Gibbs. Concurrency Control in Groupware Systems. In *ACM SIGMOD '89*, pages 399–407.

[7] Haake, Haake, and Hicks. On Merging Hypertext Networks. In *Proceedings of the ECSCW'95 Workshop on the Role of Version Control in CSCW Applications.*

[8] R. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22:4, December 1990, pages 375–408.

[9] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6:2, June 1981, pages 213–226.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:7, 1978, pages 558-565.

[11] J. Leigh and A. Johnson. Supporting Transcontinental Collaborative Work in Persistent Virtual Environments. *IEEE Computer Graphics and Applications*, 1996.

[12] J. Leigh, A. Johnson, C. Vasilakis, T. DeFanti. Multi-Perspective Collaborative Design in Persistent Networked Virtual Environments. In *IEEE 1996 Virtual Reality Annual International Symposium*, pages 253–262.

[13] J. Leigh. Personal communication, April 1997.

[14] Melly and Hall. Version Control in Microcosm. In *Proceedings of the ECSCW'95 Workshop on the Role of Version Control in CSCW Applications.*

[15] A. Prakash and M. Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 1:4, December 1994, pages 295–330.

[16] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Computer Supported Cooperative Work '96*, pages 288–297.

[17] Wieczerzycki. A Multiversion Database Model for CSCW. In *Proceedings of the ECSCW'95 Workshop on the Role of Version Control in CSCW Applications.*

```
Main:
    Initialize
    while not aborted
        if there is an input o, then
            Generate Request (o)
            Execute Request
        else
            Receive Request
            Execute Request
Initialize:
    s ← s_0
    L ← empty set
    Q ← empty set
    G ← initial interaction model
    v ← (0, 0, ..., 0)
    k ← 1
    u ← identification of local participant
Generate Request(o):
    r ← (u, k, v, o)
    Q ← Q + r
    L ← L + r
    k ← k + 1
    broadcast r to other participants
Receive Request:
    if there is a request from the network, then
        receive it as r
        Q ← Q + r
        L ← L + r
Execute Request:
    if there is r in Q satisfying Executable?(r, v), then
        choose executable request r = (j, k_j, v_j, o_j)
            Q ← Q − r
            r'' ← TranslateRequest(r, v)
            apply operation o(r'') as user j to state s
            increment jth component of v
```

Figure 3: The adOPTed algorithm for concurrency control, part I.

$Executable?(r, v)$
    logical value of $v(r) \leq v$
$TranslateRequest(r, v)$
    $(j, k_j, v_j, o_j) \leftarrow r$
    if $v_j = v$, then
        add request r to model G
        return r
    else if G contains $r\prime$ for r translated to v
        return $r\prime$
    else
        let i be such that
                $Reachable?(Decr(v, i))$ and
                $v_j[i] \leq v[i] - 1$
        $v\prime \leftarrow Decr(v, i)$
        $r_i \leftarrow Request(i, v[i])$
        $r_i\prime \leftarrow TranslateRequest(r_i, v\prime)$
        $r\prime \leftarrow TranslateRequest(r, v\prime)$
        $(r\prime\prime, r_i\prime\prime) \leftarrow tf(r\prime, r_i\prime)$
        add requests $r\prime\prime, r_i\prime\prime$ to model G
        return $r\prime\prime$
$Reachable?(v)$
    for every i in U:
        $v[i] = 0$
        or $w(Request(i, v[i])) \leq v$
$Decr(v, i)$
    return copy of v with ith component decremented
$Request(i, j)$
    return jth request from user i in L

Figure 4: The adOPTed algorithm for concurrency control, part II.

$TranslateRequest(r, v)$
    $(j, k_j, v_j, o_j) \leftarrow r$
    if $v_j = v$, then
        add request r to model G
        return r
    else if G contains $r\prime$ for r translated to v
        return $r\prime$
    else
        let i be such that
                $Reachable?(Decr(v, i))$ and
                $v_j[i] \leq v[i] - 1$
        $v\prime \leftarrow Decr(v, i)$
        $r_i \leftarrow Request(i, v[i])$
        $r_i\prime \leftarrow TranslateRequest(r_i, v\prime)$
        $r\prime \leftarrow TranslateRequest(r, v\prime)$
        $(r\prime\prime, r_i\prime\prime) \leftarrow tf(r\prime, r_i\prime)$
        add requests $r\prime\prime, r_i\prime\prime$ to model G
    *   undo r_i as user i
    *   do $r_i\prime\prime$ as user i
        return $r\prime\prime$

Figure 5: The algorithm for concurrency control with Rewriting History. (Changes from the adOPTed algorithm are marked with an asterisk.)