BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M4

# "Transaction Groups in ObServer"

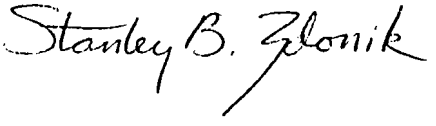
by
Mary F. Fernandez

# Transaction Groups in ObServer

Mary F. Fernandez
Brown University
Providence, RI
12 May 1989

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in
the Department of Computer Science at Brown University

Advisor
Approved by Stanley B. Zdonik



# Transaction Groups in ObServer

Mary Fernandez
Brown University
Providence, RI 02912

12 May 1989

**Abstract**

Design activities are characterized by long, interactive transactions that require mechanisms for sharing "work in progress" and for communicating about shared data. Because these transactions cooperate, the imposition of serializability as a correctness criteria for concurrency control is often too restrictive.

ObServer is an object server designed to support collaborative design applications. ObServer's concept of an object is a *byte stream* and its operations are limited to *read* and *write*. To support collaborative work, ObServer provides *communicative locking* of objects and permits the possibly non-serializable execution of cooperating transactions. In designing and implementing ObServer, it became apparent that the object server should provide provide primitives that an application or group of applications can use to concisely specify their view of correctness.

In this paper, we present a hierarchical model for specifying the logical grouping of cooperating transactions. A *transaction group (TG)* is an active entity that controls the interaction of its cooperating members and handles recovery in the face of system or member failure. Transaction groups are organized hierarchically so that non-cooperating transactions can be isolated from cooperating transactions' work. Because ObServer cannot know a priori the possible ways in which cooperating transactions will access objects, we have extended our model of concurrency control to provide user-definable *operation machines* that capture the semantics of the existing ObServer locks in addition to specifying the permissible interleavings of members' operations. Since operation machines can be defined arbitrarily, a static conflict table cannot be predefined. Thus we present algorithms for generating *machine compatibility* tables to be used instead of a lock conflict matrix. To support recovery procedures in this environment, the transitions of operation machines are annotated to add edges to *reads from* and *over write* graphs. These graphs capture the dependencies that exist between cooperating transactions and are used by the transaction group when members commit or abort.

# Contents



# List of Figures

# List of Tables

# 1 Introduction

The impetus for designing and implementing ObServer was the absence of an object server that adequately supported design activities. The goal for ObServer was to build a low level object server that could support different types of higher level design applications such as CAD/CASE tools, Hypermedia applications or interactive programming environments. Although these applications have different purposes, they share some fundamental characteristics.

- They are interactive in nature and are used by people performing design, document building or program development tasks.
- Their users tend to work in groups while designing systems or authoring documents.
- Their objects are not uniformly similar and can be related in complex and arbitrary ways (i.e., their sizes, types and interobject relationships can vary widely).

Consider an hypermedia environment in which a document may be composed of many different types of objects (e.g.,text, pictures, graphs, music, interactive programs). If multiple authors are modifying a hypermedia document, it may be useful for them to have access to each others work in progress as a way of collaborating on design. It may also be necessary to know of uncommitted changes so that related components of the document may be kept consistent (e.g.,the text referring to a graph must be modified if the data of the graph changes). It is also the case in hypermedia that different applications are needed for modifying component objects of different types (see Figure 1). However, all the objects shared by applications are

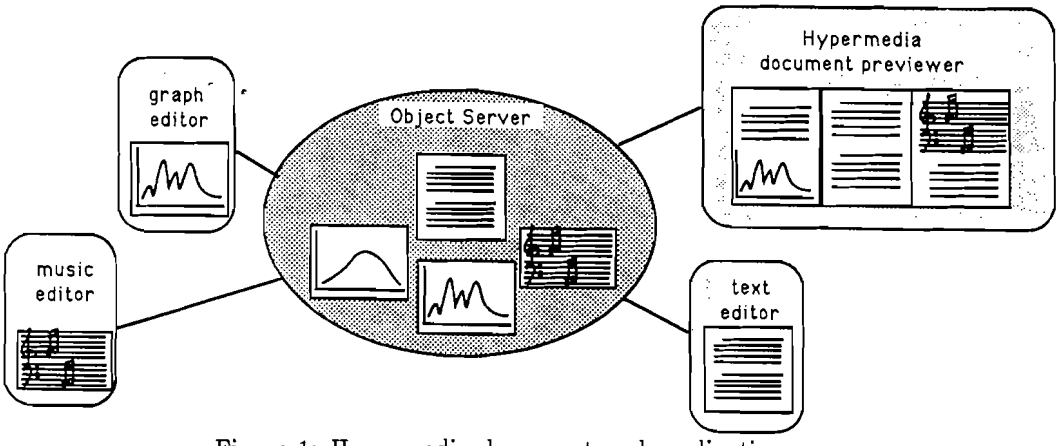



Figure 1: Hypermedia document and applications

stored in the same underlying database. For this reason, it is imperative that the underlying object server support a model in which transactions can access objects in a cooperative manner.

Our aim is to provide a transaction model that captures the hierarchical grouping of design activities and that provides user-definable concurrency control mechanisms. We do not want to omit serializability as a permissible correctness criteria, but prefer to support heterogeneous views of correctness within the same object server. Consider an environment in which groups of people are working on the design of an airplane (see Figure 2). The design effort is divided into two primary groups that are working on the engine and fuselage design and a third group that performs analysis tests on the designs. The engine group is subdivided into two more groups: one for people working on the design of the engine and another for writing maintenance manuals. The fuselage group is composed solely of design application programs. The analysis group uses engine and fuselage data as input to structural analysis programs. Depending on the interaction of the design applications, different sharing protocols may exist at each level. For instance, the structural analysis programs should not execute until the objects created by the engine and fuselage groups

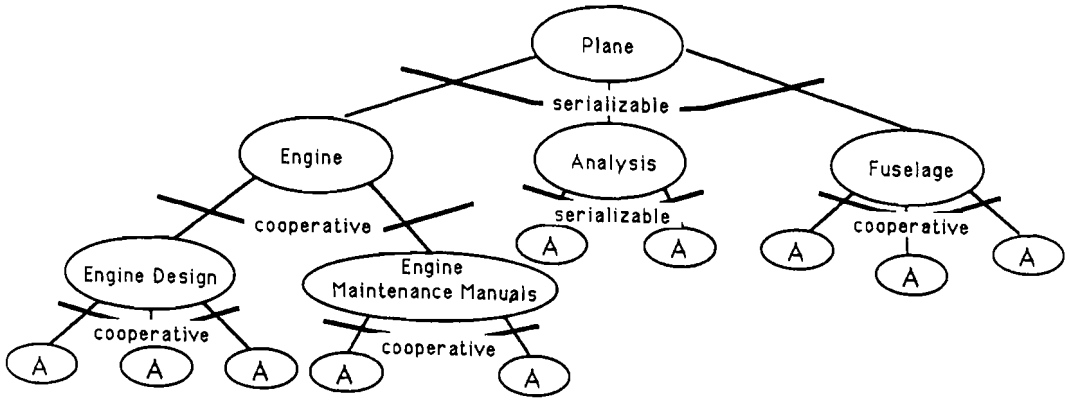



Figure 2: Groups of design applications

are complete. In this case, serializability of the groups' operations should be enforced. In the "engine design" group, design applications may be permitted to write objects simultaneously while at the "engine" group level, the "maintenance manual" group may be permitted only to read objects modified by the "engine design" group but not permitted to write them. Enforcing the same view of correctness in each group limits the possible interactions of applications and reduces concurrency. The database should support a model that enables application builders to concisely specify a group's view of correctness and provide mechanisms for mapping between nested sharing protocols so that the overall integrity of the database is preserved.

ObServer is an existing object server that provides locking and operation mechanisms to facilitate cooperation between design transactions. We have found that the current model is not expressive enough to capture the hierarchical organization of design activities. In this paper, we discuss past, present and future work on ObServer. Section 2 describes the existing model of locking and object operations supported by ObServer. We briefly discuss its merits and present an example to illuminate its weaknesses. In Section 3, we review previous database systems research on design applications and cooperative transactions and then propose our model of the transaction group in Section 4. Beginning in Section 5, we give a formal description of transaction groups and show how this model is more expressive than our current system. Finally in Section 7, we discuss areas for future research.

# 2 Current ObServer Model

Because ObServer was intended to support applications with differing type systems and object formats, ObServer supports only one type — namely a *byte-stream*. An ObServer object is a *unique identifier, byte-stream* pair. To control access to objects, ObServer provides an extended set of locks and object operations. These locks and operations support cooperation between transactions by permitting them to share "work in progress" and by notifying cooperating transactions when shared objects are updated or needed by another transaction. In this section, we describe the current implementation of ObServer and how it supports cooperative work. We also present an example that shows how the current model does not adequately control the interleaving of transactions' operations nor the interactions between cooperating groups of transactions. A complete description of ObServer's functionality is found in [HZ87].

## 2.1 ObServer Locks

A characteristic of design environments is the need for communication between cooperating transactions. A common source of interaction is the sharing of objects. If a transaction locks an object, it has control over that object until it releases its lock. In a design environment, this behavior may be too restrictive. A

transaction may need to lock an object, but would like information regarding other transactions' requests to lock the object. For this reason, ObServer supports *communicative locking* of objects. An ObServer lock is a *lock mode,communication mode* pair. The lock mode specifies whether the transaction intends to read or write the object. In addition to standard read and write lock modes, ObServer provides non-restrictive lock modes that permit multiple readers/one writer of an object or multiple readers/multiple writers. The communication mode specifies whether the transaction wants notification when another transaction is queued for a lock on an object or when another transaction has updated an object. This allows a transaction to lock an object while receiving information regarding other transactions' use of the object. The six types of lock modes and eight communication modes are described in Tables 1 and 2.

| | |
|---|---|
| *N* | Provided to allow communication on object. |
| *NR* | Non-restrictive Read. Permits other transactions non-restrictive read and write access to object. |
| *RR* | Restrictive Read. Prohibits other transactions from writing object. |
| *MW* | Multiple Write. Allows multiple transactions write access to object. |
| *NW* | Non-restrictive Write. Permits non-restrictive read access to other transactions. |
| *RW* | Restrictive Write. Provides exclusive access to an object. |

Table 1: Lock Modes

| | |
|---|---|
| *N* | No notification. Only lock modes are effective. |
| *U* | Inform transaction when the object has been updated. |
| *R* | Inform lock holder if another transaction cannot acquire a read lock. |
| *W* | Inform lock holder if another transaction cannot acquire a write lock. |
| *RW* | Inform lock holder if another transaction cannot acquire a read or write lock. |
| *UR* | Combination of U-notify and R-notify. |
| *UW* | Combination of U-notify and W-notify. |
| *URW* | Combination of U-notify and RW-notify. |

Table 2: Communication Modes

Because of the semantics of lock and communication modes, not all forty eight possible combinations result in valid locks. Table 3 specifies valid lock mode, communication mode pairs. An example of a "cooperative lock" is $NR_U$[1]. The lock mode gives the transaction read access to the object but permits other transactions to lock the object non-restrictively for writing. If another transaction does write the object, the transaction holding the $NR_U$ lock is notified that the object has changed. The reading transaction may then read the new copy of the object. Another example is $RW_{RW}$. The transaction obtains exclusive access to the object but is notified if other transactions are queued to obtain a read or write lock. This enables a transaction to release an object lock on demand.

## 2.2 ObServer Operations

ObServer provides the standard operations for obtaining locks on objects, for reading and writing objects and for releasing locks on objects. In addition to the object operations, the transaction operations commit and abort are provided (see Table 4). ObServer does not control the interleavings of operations. That

[1] Locks are written with the communication mode as a subscript to the lock mode *e.g.* $LM_{CM}$.

| Lock Modes | Communication Modes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *N* | *U* | *R* | *W* | *RW* | *UR* | *UW* | *URW* |
| *N* | I | V | I | I | I | I | I | I |
| *NR* | V | V | I | V | I | I | V | I |
| *RR* | V | I | I | V | I | I | I | I |
| *MW* | V | V | V | V | V | V | V | V |
| *NW* | V | I | V | V | V | I | I | I |
| *RW* | V | I | V | V | V | I | I | I |

Table 3: Valid Lock and Communication Mode Pairs

is, it is permissible for a transaction to read an object updated by another transaction before the writer commits. It is also possible for a transaction to update objects and then abort while the updates remain visible in the database. ObServer does not restrict the order of transactions' operations in any way. The only imposed restriction is that a transaction hold an appropriate lock at the time it requests an operation (e.g., $r_i(x)$ requires that $t_i$ hold a read type lock on object $x$ before reading $x$). Despite the apparent lack of imposed correctness criteria, this flexibility permits cooperating transactions to view objects as they are being updated by other active transactions.

| | |
|---|---|
| $r_i(x)$ | Transaction $t_i$ reads object $x$. |
| $w_i(x)$ | Transaction $t_i$ writes object $x$. |
| $rl_i(l, x)$ | Transaction $t_i$ requests a read lock $l$ on object $x$. |
| $wl_i(l, x)$ | Transaction $t_i$ requests a write lock $l$ on object $x$. |
| $ru_i(x)$ | Transaction $t_i$ releases its read lock on object $x$. |
| $wu_i(x)$ | Transaction $t_i$ releases its write lock on object $x$. |
| $c_i$ | Transaction $t_i$ commits. All locks held by $t_i$ are released. |
| $a_i$ | Transaction $t_i$ aborts. All locks held by $t_i$ are released. |

Table 4: Operations on Objects and Transactions

The following example taken from [FZ89] illustrates the use of ObServer locks and object operations between two cooperating transactions (see Figure 3). Consider two transactions $t_1$ and $t_2$ which correspond to two design applications. The applications are used by two people collaborating on the design of an airplane jet engine. $t_1$ is updating the engine fuselage *of* and $t_2$ is modifying the turbine *ot*. $t1$ is also reading *ot* and wants to be notified if the object is updated. $t1$ requests a $NR_U$ lock on *ot*. When $t_2$ updates *ot* to *ot'*, $t_1$ is notified of the update. $t_1$ may then reread *ot* from the server.

## 2.3 Cooperation Revisited

We now introduce a third transaction into our example to demonstrate how ObServer fails to insulate non-cooperating transactions from the effects of cooperating transactions. $t_3$ is a documentation monitor. The application executing $t_3$ rereads *ot* whenever it is updated so that the engine turbine documentation may be updated to reflect design changes. $t_3$ holds a $NR_U$ lock on *ot*. When *ot* is updated by $t_2$, the server notifies both $t_1$ and $t_3$ that *ot* has been updated (see Figure 4). When $t_1$ and $t_2$ are making design changes, $t_3$ may be triggered to execute prematurely and could use an inconsistent or incomplete version of *ot*. $t_1$ and $t_2$

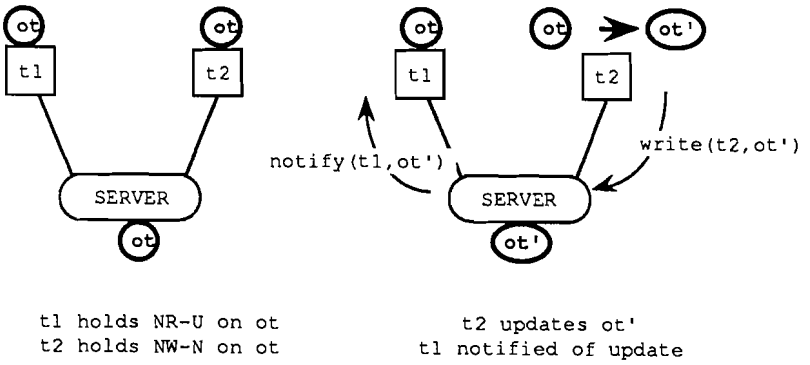



Figure 3: Use of update notification

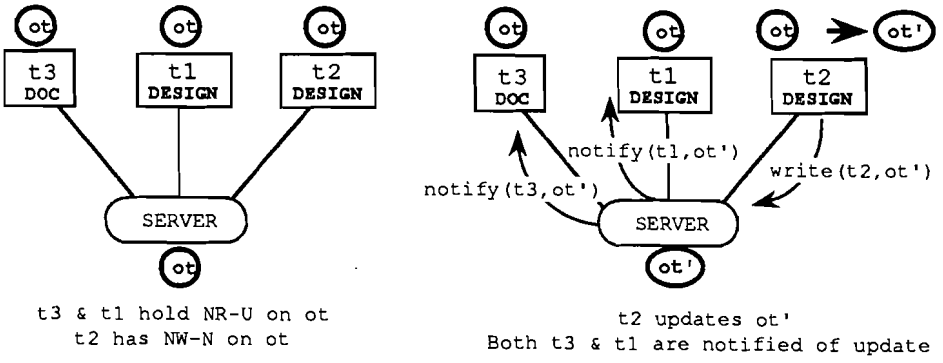



Figure 4: Notification of non-cooperating transaction $t_3$

comprise a design group whose behavior should be isolated from other non-cooperating transactions.

## 2.4 ObServer Recovery

The recovery method chosen for a database must handle failures cause by external events, such as site crashes and network partitions, as well as internal events, such as transaction aborts and database failures. Regardless of the cause, the chosen recovery method must preserve the permanence and consistency of the data. It is recognized that recovery interacts subtlely with the chosen concurrency control method [Wei89]. For instance, in a database that enforces serializability via two phase locking (2PL), cascading aborts can be avoided if the system uses *strict 2PL* [BHG87]. However, in an unconventional system like ObServer, where possibly non-serializable histories are permitted, system-instigated aborts of transactions are not acceptable. In ObServer, a transaction is not aborted even if it read from a transaction that subsequently aborts. This philosophy reflects the fact that design transactions are interactive and autonomous. Aborting an interactive transaction is not permissible if a person's work will be sacrificed. Because of this, ObServer transactions are not necessarily atomic. Updates to objects made by an aborted transaction may remain in the database if another active transaction is dependent on that change. When referring to ObServer transactions, the terms "commit" and "abort" connote "successful completion" and "unsuccessful completion" of a transaction's computation instead of atomic completion or failure.

Just as when a transaction aborts unexpectedly, when a database or system failure occurs the system must decide which transactions to redo or undo. In a system that performs *selective redo* during recovery procedures, transactions that were uncommitted at the time of the crash are undone and committed transactions are redone. In the current ObServer implementation, we ascribe to a *redo-only* crash recovery scheme. That is, we "repeat history" since it is possible that committed transactions read from uncommitted transactions and are thus dependent upon their changes. This is similar to the scheme used in ARIES [MHL+89].

For example, if transaction $t_1$ shares its work with other transactions by updating an object and then the system crashes before $t_1$ commits but after transactions that read from $t_1$ commit, potential anomalies may exist if $t_1$ is not redone. The server, however, cannot unilaterally undo all transactions that read from $t_1$ since cascading undos can ensue. The dependencies that existed between $t_1$ and transactions that read from it should maintained so that recovery mechanisms can choose a combined strategy of redoing and undoing transactions. We would like then to provide a recovery model that captures the dependencies created between transactions sharing work-in-progress so that a transaction abort forces the database to either generate warnings, delay committal of dependent transactions or wait for a compensating transaction to reconcile potential inconsistencies.

### 2.5 Weaknesses of ObServer's Model

ObServer's extended lock set and redo-only recovery scheme support cooperative transactions by increasing concurrency and by permitting them to share work-in-progress. However, we have identified three problem areas that the current ObServer does not address. These are

- non-cooperating transactions are not isolated from the interaction of cooperating transactions,
- the interleavings of cooperating transactions' operations are not controlled,
- the recovery model does not recognize the dependencies that exist between cooperating transactions that share data.

Allowing cooperation among a privileged set of transactions should not subject all transactions to the possibly non-serializable behavior of the group, nor make them privy to a group's uncommitted changes. For those isolated transactions that are cooperating, the system should support user-definable orderings of transactions' operations and a reliable recovery mechanism that recognizes dependencies between transactions. In the next section, we review other research in the area of cooperative transaction models and then propose our mechanism, the *transaction group*, for specifying and controlling the interaction of design transactions.

## 3 Related Research

The need for a transaction model that supports cooperative design activities was recognized during the development of complex, integrated CAD and CASE tools. Traditional databases supported a model of short duration, serializable transactions. The application of this model to the long lived, interactive and distributed transactions of design tools was difficult. Thus the development of an extended transaction model to support both types of database transactions was needed.

A few models whose development was driven by real world design problems share a number of similarities [KKB87, K+84, K+85, LP83]. They all provide mechanisms for organizing transactions hierarchically and assume that an area exists at each level for caching shared objects. These areas can be permanent databases or temporary areas for storing object and lock information. All the models limit the depth of the transaction hierarchy to three layers, consisting of *public*, *group*[K+85, R+88] or *semi-public*[K+84], and *user* or *private* databases. The public layer contains objects that may be accessed by any application or group transaction. The group layer is restricted to a set of privileged users or transactions that are sharing objects. The user layer is accessible only by its owner. In both [K+85] and [KKB87], transactions may only "belong" to one group or have access to one semi-public database. The structure of the database resembles a tree with the public database at the root, the semi-public databases as children of the root and the user transactions as children of the semi-public databases. In [K+84], transactions are permitted to "check out" objects from any other transaction thus producing a DAG of cooperating transactions.

Although the proposed models could support a multilevel hierarchy of group or semi-private databases, all maintain that a static three level model is adequate. They also assume that intergroup cooperation is unnecessary and thus enforce serializability of the group transactions. They do not permit groups to share objects as member transactions do. In [KKB87], projects must abide by two phase locking when requesting objects from the public database and in [K+85], group transactions are guaranteed to be serializable. This assumption limits the flexibility of group behavior. It should be possible to specify sharing protocols for groups as well as for transactions. The underlying database should thus support multiple levels of groups as well as heterogeneous sharing protocols among the groups.

In [Lyn83], an abstract model for specifying multiple levels of transactions is presented. Lynch provides a framework in which the correctness criteria of serializability is relaxed and is extended to include a broader set of allowable interleavings. Her model does not require that groups interleave their operations in a strictly serializable order nor does it restrict the depth of transaction groupings. She assumes however that groups of transactions can predeclare a partial ordering of their operations based on the semantics of the application. These partial orders are used to determine the correctness of multiple levels of interleavings. In her model, consistency constraints are not associated with the data since she argues that it is complicated to state these constraints. However, in an object-oriented database, consistency contraints can be specified by the methods implemented for an object and by controlling the interleavings of the methods [Ska88].

The models derived from practice recognize the need for an extended lock set to support the cooperative behavior of design transactions. In addition to standard read and write locks, [LP83] proposes a lock for writing and deleting an object and another for reading while others write. These locks are similar to ObServer's non-restrictive read and write locks. Like the current ObServer, the forementioned models do not provide user-definable access patterns for objects. Enabling the application designer to specify the permissible patterns of object use among cooperating transactions increases the flexibility of the transaction model.

A characteristic of design activities not addressed by these approaches is communication. Cooperative behavior requires primitives for communicating about shared objects and for defining extensible access patterns. Because the transactions cooperate by sharing objects, the mechanism for communicating should provide information on the use of shared objects. For this reason, communicative locks are extremely useful. As a lock mode guarantees access to an object, a communication mode guarantees that the lock holder is notified of requests for the object. In this way, cooperation is inherently supported by the locking model. None of the previously mentioned approaches provide primitives for communicating about shared objects.

# 4 Transaction Groups

We have already mentioned some important characteristics of design applications such as autonomy, interactivity, heterogeneity and clustering. Although people work cooperatively on a design activity, they are autonomous. They do not necessarily use the same application programs, work on the same machine, or follow the same schedule. The applications they use in design are interactive and are not necessarily homogeneous (e.g., text, forms and graphics editors may be used to constuct a single technical manual). And lastly, the designers cluster their work into groups dynamically. The entity that binds together different designers' work is the shared data and thus the locus of sharing in these environments is the database. We posit that by providing an extensible model of transactions at the object server level, the clustering and interaction of design applications can be more easily defined and controlled than in systems that ascribe to a strictly serializable transaction model.

We propose a hierarchical model for specifying the logical grouping of cooperating transactions in an object server. A *transaction group (TG)* is an active entity that controls the interaction of its cooperating members and handles recovery in the face of system or member failure. A *transaction group member (M)* is

an individual transaction or another transaction group. The TG localizes the interaction of its members by adhering to an internal protocol. As a member of another TG, a TG must translate its internal protocol into an external protocol compatible with its parent's sharing patterns. In this way, each level in the TG hierarchy adheres to its own view of correctness. This permits encapsulated groups of non-serializable transactions to coexist with, and be members of, other serializable groups.

To its members, a TG appears to be the only database server responsible for storing objects, controlling member access to objects, notifying members of object use and handling recovery. The TG, however, may be a member of another TG from which it must request object access and from which it receives objects for local caching. The TG hierarchy is a tree with the root TG being the permanent database. We use the familial vocabulary to identify members (i.e.,the group from which a member requests object access is its *parent* and any members which request objects from it are its *children*).

Because we can not determine the clustering of transactions a priori, the TG hierarchy can be constructed dynamically. A group is instantiated from a *group template* ($\mathcal{GT} =< \mathcal{I}, \mathcal{E} >$) which specifies the behavior and structure of the group. The template consists of internal ($\mathcal{I}$) and external ($\mathcal{E}$) protocols. The internal protocol includes information such as

- number and identities of members,
- a concurrency control mechanism to control the interleaving of member operations,
- size of the TG's internal object cache and
- rules for handling recovery due to member failure.

The external protocol captures how the group translates its internal activity into interactions with its parent. This includes information such as

- rules for handling object messages received from parent,
- mappings between the internal concurrency control mechanism and that of the parent group,
- rules for returning objects to the parent from an internal object cache.

An instance of a TG, $\mathcal{G} =< \mathcal{N}, \mathcal{GT}, \mathcal{P} >$, is an active object server constructed from a group template $\mathcal{GT}$. Each TG is named $\mathcal{N}$ and has a unique parent group $\mathcal{P}$, unless it is the root database in which case it has no parent. In the following sections, we describe some important internal protocols such as controlling members' access to objects and handling member failure, and external protocols such as mapping between the internal activities of members and external interaction with the TG's parent and siblings.

# 5 Controlling Access to Objects

In this section, we present a user-definable concurrency control mechanism called an *operation machine* for controlling the interleaving of members' operations. We describe how compatibility between operation machines is computed and how levels of operation machines are mapped in the TG hierarchy. The internal protocol of a TG includes a set of operation machines usable by group members and its external protocol includes a set that it uses when requesting objects from its parent group.

The current ObServer controls object use by granting communicative locks to transactions, but it does not control the orderings of operations invoked by transactions. For example, multiple writer locks permit transactions to over write each others changes. A more controlled use of multiple writer locks would require multiple writers to modify an object in a round-robin fashion. Initially, the set of lock and communication modes provided by ObServer seemed adequate to express all possible cooperative usage patterns needed by

transactions. It is now apparent that this set is not exhaustive and that a more expressive model of locking is necessary to facilitate as well as control cooperative transactions. Because we can not specify a priori all possible patterns used by transactions, our new model must be extensible so that new operation patterns can be defined by the application builder.

The method we propose is an adaptation of Skarra's model for specifying concurrency control in an object-oriented database [Ska88]. In lieu of serializability, a transaction group's criteria for determining the correctness of an operation history is based upon operation patterns and conflict specifications. Permissible patterns of object operations invoked by transaction group members are specified formally using augmented finite state automata[Ska89]. We call these automata *operation machines*. Machine transitions are annotated with predicates that check for operation conflict and lists of actions to execute when the transition arc is traversed. Based on member identify and arguments to the operation, the predicates determine whether an operation is permissible, conflicts with other operations or should be queued. Actions may include notifying members of interesting events or adding recovery information to group recovery graphs. Because of their expressivity, operation machines encompass the semantics of existing ObServer locks while specifying permissible orderings of operations by group members. In the next section, we describe Skarra's model and our simplifications for use in ObServer's transaction groups.

## 5.1 Skarra's Operation Machines

The framework in which we describe operation machines is an object-oriented database in which each object is an instance of a predefined type. A type specification includes internal, or hidden, information regarding the object format and external, or public, information describing the methods for accessing the object. For example, the type **file** might include methods to **open, read, write** and **close** an object of type **file**. The type specification does not describe the permissible orderings of these operations (e.g., **open** must precede **read** or **write**) nor does it describe how group members might be able to share a **file** object (e.g., two transactions may read concurrently but only one transaction may write).

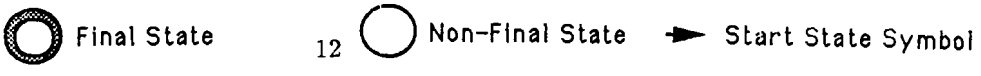
An *operation machine template* augments a type definition by specifying the permissible orderings of object operations by transaction group members. Each operation machine is an active instance of a machine template. A machine template ($\mathcal{MT}$) is a tuple defined by:

$$\mathcal{MT} =< K, \Sigma, \Delta, s, F, L, O, M, B_o, B_m > \quad (1)$$

$K$ is a finite set of states,
$\Sigma$ is an alphabet,
$\Delta$ is a transition function from $K \times \Sigma \rightarrow K$
$s \in K$ is the initial state,
$F \subseteq K$ is the set of final states,
$L$ is a set of local variables,
$O$ is a set of object names,
$M$ is a set of member names,
$B_o$ is a binding function from object identifiers to object names,
$B_m$ is a binding function from member identifiers to member names.

In addition to expressing the permissible orderings of object operations, operation machines capture the "completeness" of a sequence of operations by distinguishing between final and non-final states. If a machine is in a final state, it has either completed a required operation or has accessed the object in a permissible order. For example, in Figure 5[2] the member bound to $m$ must execute the **close** operation before committing. If member $m$ then commits, the pattern for accessing the **file** object is complete. However

[2]Legend for operation machine diagrams:



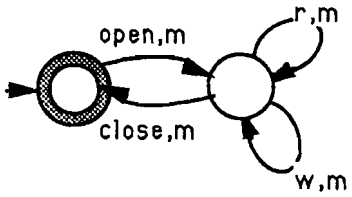



Figure 5: Operation machine for type **file**.

if the machine is in a non-final state, either the operation sequence is invalid or incomplete. In this instance, the member would not be permitted to commit.

### 5.1.1 Instantiating Operation Machines

When a member transaction invokes an operation on an object for the first time, the transaction group instantiates an operation machine from an appropriate template. The template is chosen from a set of templates that are compatible with the sharing protocol defined for the group. The binding functions $B_o$ and $B_m$ are used to bind object and member identifiers to the object and member names used in the machine. Because many members may be "sharing" the object by invoking operations concurrently, multiple machines may be activated for a single object. The transaction group (TG) executes these machines in parallel. When an operation is requested by a member on an object, the TG feeds the operation to the machine(s) activated for the object. Each machine returns a response to the group indicating whether the operation conflicts, is permissible or should be queued. The TG determines from the responses whether the operation should proceed, be queued or be refused. If all machines return proceed, the operation is executed. If the responses include proceed and queue, the operation is queued. Should any refuse responses be returned, the operation is not executed and the member is notified of an operation conflict.

## 5.2 ObServer's Operation Machines

Because ObServer does not support a complex type system and therefore does not permit arbitrary operations on objects, specifying operation machines in the ObServer environment is simplified. As previously discussed, the current ObServer provides a predefined set of locks but cannot express explicit orderings of read-write sequences. Operation machines can capture the existing semantics of ObServer locks as well as allow an application builder to define his own "lock" or object access patterns. We now define the operation machine template as used by ObServer.

A machine template ($\mathcal{MT}'$) is a tuple defined by:

$$\mathcal{MT}' =< K, \Sigma, \Delta, s, F > \tag{2}$$

$K$ is a finite set of states,
$\Sigma$ is an alphabet,
$\Delta$ is a transition function from $K \times \Sigma \rightarrow K$
$s \in K$ is the initial state,
$F \subseteq K$ is the set of final states,

In Skarra's model, multiple objects may be referenced in the same machine. This enables the type designer to specify intertype operation dependencies. Because ObServer is unaware of interobject relationships, an instance of a machine template binds explicitly to one object. We also restrict the member set to include an explicitly bound member ($m$) and all other members not equal to $m$ ($\bar{m}$). This allows us to express which operations are permissible for a particular member and which operations are permissible or conflicting

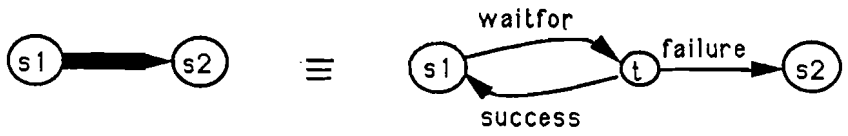



Figure 6: Synchronization submachine

for all other members. Since operations in ObServer are limited to read and write, transitions are made on *operation, member* pairs from the cross product $\{r, w\} \times \{m, \bar{m}\}$. A symbol $\sigma$ in the alphabet $\Sigma$ is a quadruple $\sigma =< \mathcal{O}, \mathcal{M}, \mathcal{P}, \mathcal{A} >$ where

$\mathcal{O}$ is an operation $\in \{r, w\}$,
$\mathcal{M}$ is a member $\in \{m, \bar{m}\}$,
where $m \equiv$ a member identifier and $\bar{m} \equiv$ any member except $m$.
$\mathcal{P}$ is $\in \{p, r, q\}$ where $p$ is proceed, $r$ is refuse and $q$ is queue.
$\mathcal{A}$ is a list of actions to perform if the operation is executed.

By permitting an arbitrary list of actions to be associated with each transition, we allow the user to define interesting events. ObServer communication modes can be modeled as lists of events that notify a member when an object is updated or if another member is queued for a lock. When a group member requests a machine to be instantiated, we assume that it is prepared to accept any messages from the group that can be generated by the action lists of the machine. In the example machines that follow, the types of actions included in action lists are :

| | |
|---|---|
| *notify(m, e)* | Notify member $m$ of interesting event $e$. Events include *update* of an object, *writeq* when a write operation is queued and *readq* when a read operation is queued. |
| *waitfor($m_1$, op, $m_2$)* | Operation *op* is queued for member $m_1$ by member $m_2$. |
| *set(p, v)* | Set value of predicate $p$ to value $v$. |
| *add(g, e)* | Add edge $e$ to graph $g$. |
| *delete(g, e)* | Delete edge $e$ from graph $g$. |

The *set*, *add* and *delete* operations are used for constructing graphs used during recovery operations (see Section 6).

### 5.2.1 Synchronization Submachines

The default action list for any transition that queues an operation includes the action *waitfor($m_1$, op, $m_2$)*. Each *waitfor* operation adds an edge to a *waits-for* graph maintained by the transaction group. This graph can be used to identify potential deadlocks between members waiting for access to each others objects. Edges are FIFO ordered so that queued operations are dequeued in the order that they were requested. The *waitfor* action returns either *success* or *failure* indicating that the operation was queued successfully or an exception occurred. An exception occurs if the TG decides to override the machine transition that queues the operation and lets the operation proceed. If this occurs, the semantics of the operation machine have been violated and the machine must proceed to a dead state. This mechanism is represented in Figure 6. The synchronization submachine is used in the examples that follow wherever a transition indicates that an operation should be queued.

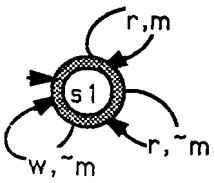
### 5.2.2 Examples of Operation Machines

To illustrate the use of operation machines, we give examples of templates that capture the semantics of existing ObServer locks and examples of operation patterns that the current ObServer cannot express.

An ObServer lock mode is modeled in an operation machine as two types of transitions, one type for operations relevant to the lock holder ($m$) and one type for operations relevant to all other transactions ($\bar{m}$) in the TG. For each operation permitted by the holder of the lock, a transition is of the form $< \mathcal{O}, m, p, \mathcal{A} > \in \Delta$. For each operation prohibited by the holder of the lock, a transition is of the form $< \mathcal{O}, m, r, \mathcal{A} > \in \Delta$. If a transition is omitted for a particular *operation, member* pair, the default predicate value is refuse. For each operation that does not conflict with the lock (i.e., permitted to execute by $\bar{m}$), a transition is of the form $< \mathcal{O}, \bar{m}, p, \mathcal{A} > \in \Delta$, and for each operation that does conflict with the lock (i.e., cannot be executed by $\bar{m}$), a transition is of the form $< \mathcal{O}, \bar{m}, q, \mathcal{A} > \in \Delta$. Queuing an operation places a tuple $< m, p, o, m' >$, where $m$ is the member requesting the operation, $p$ is the operation, $o$ is the object and $m'$ is the member bound to the machine that queues the request, on a queue of pending operations. Queuing the operation $o$ does not suspend the machine bound to $(o, m)$ since operations other than $p$ may be able to proceed for member $m$ on object $o$. This is different than Skarra's model in which any machine that has an opertion queued by another machine is suspended. We do not discuss here how pending operations are dequeued and resubmitted to active machines. This problem requires additional investigation.

ObServer communication modes are modeled as actions in the action list of each transition. Update notification is specified as a *notify(m, update)* action and queuing of read or write locks is specified as *notify(m, readq)* or *notify(m, writeq)*. Because each machine is bound explicitly to one object, the *notify* action implicitly includes the object identifier in the message sent to the member.

In the following examples, a figure representing the ObServer lock is drawn on the left[3] and the corresponding transitions are on the right. In the definition of action lists, $\epsilon$ is the empty list and lists of actions are bracketed by [ and ]. Figure 7 defines an $NR_U$ ObServer lock. The machine permits the lock holder to read and any other transaction to read or write. If another transaction does write, the member is notified of the update. An $RR_W$ lock is modeled in Figure 8. The lock holder (bound to member $m$) and all other members are permitted to read, but writes are queued. Whenever a write is queued, the member is notified of the pending operation.

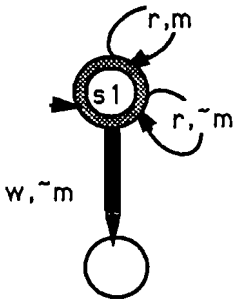



$$\begin{aligned}\Delta \;=\; \{&< s1, < r, m, p, \epsilon >, s1 >,\\ &< s1, < w, m, r, \epsilon >, s1 >,\\ &< s1, < r, \bar{m}, p, \epsilon >, s1 >,\\ &< s1, < w, \bar{m}, p, [\text{notify}(m, \text{update})] >, s1 >\}\end{aligned}$$

Figure 7: Non-restrictive read lock with update notify.

The examples presented in Section 2 demonstrated that static locking cannot control the ordering of operations. With operation machines, we can capture the semantics of locks as well as permissible orderings of operations by TG members. In Figure 9, the standard multiple writer lock is extended so that each write must be preceded by a read of the most recent update. The start state of this machine is $s1$. If the member bound to this machine reads the object, the machine transitions to state $s2$. While in $s2$, $m$ can read or write the object and other members may read. However, if another member writes the object, the machine

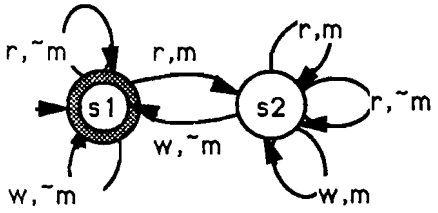
[3] In the diagrams, ~m represents $\bar{m}$




$$\begin{aligned}\Delta \;=\; \{&< s1, < r, m, p, \epsilon >, s1 >,\\ &< s1, < w, m, r, \epsilon >, s1 >,\\ &< s1, < r, \bar{m}, p, \epsilon >, s1 >,\\ &< s1, < w, \bar{m}, q, [\text{notify}(m, \text{writeq})] >, s1 >\}\end{aligned}$$

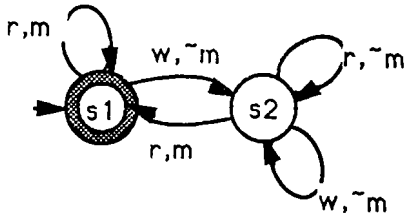
Figure 8: Restrictive read lock with write notify.

transitions to $s1$. In this state, $m$ is not permitted to write because it has not read the most recently modified copy of the object. This forces cooperating members to write the object in a round-robin fashion. Figure 10 extends a non-restrictive reader lock so that the member must read the most recent update of the object before committing. Any time the object is written, the machine transitions to $s2$. The machine will be in the final state only if $m$ has read the most recent write of the object.




$$\begin{aligned}\Delta \;=\; \{&< s1, < r, \bar{m}, p, \epsilon >, s1 >,\\ &< s1, < w, \bar{m}, p, \epsilon >, s1 >,\\ &< s1, < w, m, r, \epsilon >, s1 >,\\ &< s1, < r, m, p, \epsilon >, s2 >,\\ &< s2, < r, m, p, \epsilon >, s2 >,\\ &< s2, < r, \bar{m}, p, \epsilon >, s2 >,\\ &< s2, < w, m, p, \epsilon >, s2 >,\\ &< s2, < w, \bar{m}, p, \epsilon >, s1 >\}\end{aligned}$$

Figure 9: Round-robin multiple writers machine.




$$\begin{aligned}\Delta \;=\; \{&< s1, < r, m, p, \epsilon >, s1 >,\\ &< s1, < w, m, r, \epsilon >, s1 >,\\ &< s1, < w, \bar{m}, q, [\text{notify}(m, \text{update})] >, s2 >,\\ &< s2, < r, \bar{m}, p, \epsilon >, s2 >,\\ &< s2. < w, \bar{m}, P, [\text{notify}(m, \text{update})] >, s2 >,\\ &< s2, < r, m, p, \epsilon >, s1 >\}\end{aligned}$$

Figure 10: Machine that reads last write before commit.

## 5.3 Machine Compatibility

By providing user-definable operation machines in lieu of predefined locks, we can no longer use a static lock conflict table to determine whether two locks are compatible. Currently, the lock conflict table is a matrix formed by the cross product of the ObServer lock modes (see Table 1). The matrix entries are either *grant* or *queue* which indicate whether a requested lock should be granted or queued based on the strongest lock held by a transaction. For example, a non-restrictive read lock requested on object $x$ is queued if some transaction already holds a restrictive write lock on $x$. A non-restrictive read lock would be granted if the strongest lock held was non-restrictive write.

In our new model, locks are replaced with user-definable operation machines. These machines are executed in parallel by the transaction group. If we ignore potential conflict, priority cannot be given to existing machines over new machines. For example, if a machine that represents a $NR_U$ lock is active for a member $m$ and member $m'$ requests that a $RW_N$ machine be instantiated, the newly instantiated machine will restrict the existing machine by queuing (or refusing) any reads by $m$. Effectively, the new machine delays or prohibits rights previously granted to $m$. The new model should preserve priority of active machines over new machines as ObServer guarantees priority of granted locks over requested locks.

We assume that there exists a library $\mathcal{L}$ of machine templates accessible by all transaction groups. We are interested in an algorithm that can determine whether a machine template $t$ requested by a member $m$ on an object $x$ is compatible with all active machine instances bound to $x$ in the TG. We assume that no active machines for object $x$ are already bound to $m$ at the time of the request. We also assume that the TG has adequate rights to the object to execute a machine on behalf of the member. In Section 5.4 we discuss how a TG acquires adequate privileges to an object from its parent group through its external protocol.

### 5.3.1 Comparing Machine Templates

Assume that one active machine instance $i$ is bound to $x$ and that member $m$ is requesting that machine template $t$ be instantiated for object $x$. A simplistic algorithm for computing compatibility would compare the template $t$ to the template for $i$ ($t_i$). If there is any operation permitted to execute in $t_i$ by member $m$ ($\sigma_i =< o, m, p, A >$) that is queued or refused for $\bar{m}$ in $t$ ($\sigma =< o, \bar{m}, q$ or $r, A >$), then we know that the new template can potentially restrict the functionality of the existing machine. In this instance, $t$ and $i$ are incompatible since instantiating $t$ would give it higher priority over the existing machine $i$. And conversely, if there is any operation permitted to execute in $t$ by member $m$ ($\sigma =< o, m, p, A >$) that is queued or refused for $\bar{m}$ in $t_i$ ($\sigma_i =< o, \bar{m}, q$ or $r, A >$), then we know that the existing machine can potentially restrict the functionality of the new machine.

This definition, however, is too restrictive as shown in the following example. Consider the machine templates $t_a$ and $t_b$ in Figure 11. Template $t_a$ is a machine that affectively functions as a restrictive write lock followed by a non-restrictive write lock (i.e.,the first submachine has exclusive access to the object for some time. When the machine transitions on $\sigma$, it moves into a submachine that permits multiple readers/one writer). Template $t_b$ is a non-restrictive read lock. If we compare templates $t_a$ and $t_b$ statically (see Figure 12(a)), the read operation permitted in $t_b$ can be either queued or refused in the $RW$ submachine of $t_a$ but can proceed in the $NW$ submachine. By the above definition of compatibility, template $t_b$ could not be instantiated since it conflicts with $t_a$. However, if we consider the instantiated machine $i$ of template $t_a$, we can determine whether its current state is either in the first or second submachine. Consider Figure 12(b) in which $i$ has transitioned on $\sigma$ to the $NW$ submachine. In this case, it is irrelevant that template $t_b$ conflicts with the $RW$ submachine since $i$ is in a submachine that no longer conflicts with $t_b$ and no path exists from the $NW$ submachine back to the $RW$ submachine.


Figure 11: Machine templates $t_a$ and $t_b$

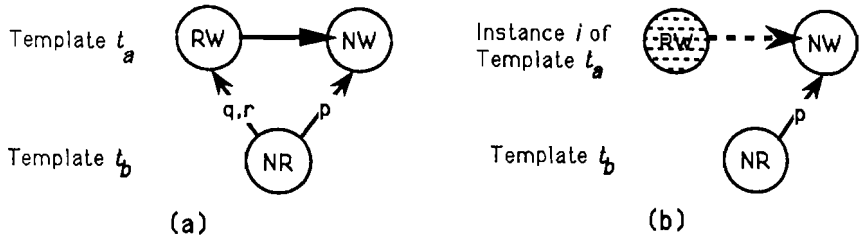



Figure 12: Compatibility of templates $t_a$ and $t_b$

### 5.3.2 Submachine Compatibility

We refine our definition of machine compatibility to consider the submachines of each operation machine template. We are interested in determining the compatibility of a template $t$ with the current submachine $s$ of an active machine instance $i$ and with all submachines *reachable* from $s$. We define a submachine of a machine template to be a set of states $K' \in K$ and transitions $\Delta' \in \Delta$ in which there are zero or more transitions from any state $p \in K'$ to $q \in K'$ and from $q$ to $p$.

**Definition 5.1** *A submachine* $\mathcal{MT}' =< K', \Sigma, \Delta' >$ *of a machine template* $\mathcal{MT}$ *is a set of states and transitions such that* $K' \subseteq K$, $\Delta' \subseteq \Delta$ *and* $(p, v) \overset{*}{\vdash} (q, w), (q, w') \overset{*}{\vdash} (p, v'), p, q \in K'$.

**Definition 5.2** *Submachine* $\mathcal{MT}''$ *is reachable from submachine* $\mathcal{MT}'$ *if there exists one or more transitions in* $\Delta$ *such that* $(p, v) \overset{*}{\vdash} (q, w), p \in K', q \in K''$.

Our new definition of compatibility uses the definitions for submachines and submachine reachability to determine the compatibility of a template with an active instance of an operation machine. We take an optimistic approach to compatibility by assuming that even if an operation is queued or refused in some state of a submachine, as long as it can proceed from one or more states in the submachine it will eventually be able to proceed. This assumption relies on the fact that in a submachine, a path exists from every state to every other state.

The following definitions capture the three possible relationships between machine templates. Informally, two templates are *compatible* if the operations that can proceed for $m$ in one template can proceed for $\bar{m}$ in each submachine of the other template. This does not mean that some submachine will not queue or refuse the operation for $\bar{m}$ but that optmistically, the machine instance will always be in a submachine that includes a state from which the operation can proceed. Two templates are *queue compatible* if the operations that can proceed for $m$ in one template can proceed or queue for $\bar{m}$ in each submachine of the other template. This means that at least one submachine of a template includes no states that let the operation proceed unconditionally, but does include one state that queues the operation. Queue compatibility implies that there is some submachine in one template that may force the other to wait. Two templates are *incompatible* if there is some submachine $s$ in a template in which every state refuses an operation that may proceed in the other template. In this case, it is possible that an instance of the template executing in $s$ will completely block out other members' operations. Next, we provide a formal definition of template compatibility that will be used by the algorithm for computing compatibility.

**Definition 5.3** *Machine template $t$ and machine instance $i'$, created from template $t'$ and currently in submachine $s'$, are* **compatible** *if*

1. *for any operation $o$ that can proceed for member $m$ in $t$, $o$ can proceed for $\bar{m}$ from one or more states in $s'$ and from one or more states in every submachine reachable from $s'$ AND,*
2. *for any operation $o$ that can proceed for member $m$ in $s'$ and all submachines reachable from $s'$, $o$ can proceed for $\bar{m}$ from one or more states in every submachine of $t$.*

**Definition 5.4** *Machine template $t$ and machine instance $i$ are* **queue compatible** *if*

1. *for any operation $o$ that can proceed for member $m$ in $t$, $o$ is queued for $\bar{m}$ in one or more states of $s'$ or in one or more states in any submachine reachable from $s'$ AND,*
2. *for any operation $o$ that can proceed for member $m$ in $s'$ and all submachines reachable from $s'$, $o$ is queued for $\bar{m}$ from one or more states in every submachine of $t$.*

**Definition 5.5** *Machine template $t$ and machine instance $i$ are* **incompatible** *if*

1. *for any operation $o$ that can proceed for member $m$ in $t$, $o$ cannot proceed or queue for $\bar{m}$ in any state in $s'$ or in any state in some submachine reachable from $s'$ OR,*
2. *for any operation $o$ that can proceed for member $m$ in $s'$ and all submachines reachable from $s'$, $o$ cannot proceed or queue for $\bar{m}$ in any state in any submachine of $t$.*

### 5.3.3 Computing Machine Compatibility

To compute compatibility, machine templates are viewed as directed labeled graphs. We define a one-to-one mapping between a machine template $\mathcal{MT}$ and a directed graph $G$ on a set of vertices $V$, labels $L$ and edges $E \subseteq V \times L \times V$. Let $\mathcal{MT}$ be defined as in Equation 2 and $G = (V, E)$ then

$V \equiv K$ is the set of vertices,
$L \equiv \Sigma$ are the labels of the edges,
$E = \{< p, \sigma, q >: p, q \in K, \sigma \in \Sigma, < p, \sigma, q > \in \Delta\}$ are the edges.

By redefining a machine template as a directed graph, we can easily compute the *strongly connected components*(SCCs) of $G$. An SCC of a graph $G$ is a subgraph $G_i = (V_i, E_i)$ such that for every pair of vertices $v, w \in V_i$ there is a path from $v$ to $w$ and a path from $w$ to $v$ [AHU74]. The strongly connected components set of a graph SCCS($G$) is the set of all such subgraphs $G_i$. SCCS($G$) partitions $G$ into subgraphs which represent the submachines of the machine template $\mathcal{MT}$.

Given a submachine $s$, we also need to compute the set of submachines reachable from $s$. Given the SCCS($G$) defined above, we construct another graph in which the vertex set is exactly the set of subgraphs in SCCS($G$). Edges exist between the new vertices if one or more edges exist between two SCC$s \in$ SCCS($G$). We call this compressed image of the machine template an *SCC-graph.*

**Definition 5.6** *An SCC-Graph $G_{scc} = (V_{scc}, E_{scc})$ of a directed graph $G = (V, E)$ is a directed graph in which each strongly connected component of $G$ is represented by a vertex in $G_{scc}$ An edge exists in $G_{scc}$ if there exists one or more edges between two distinct strongly connected components of $G$.*

We can construct $G_{scc}$ from $G$ as follows:

- for each graph $G_i = (V_i, E_i) \in$ SCCS($G$), add $v_i \equiv V_i$ to $V_{scc}$.

This constructs the vertex set of $G_{scc}$. Then construct the set of edges:

- for each vertex $v_i \in V_{scc}$, add $e = (v_i, v_j)$ to $E_{scc}$ if $\exists v, w$ such that $(v, w) \in V, v \in V_i, w \in V_j$ and $i \neq j$.

The graph in Figure 13(a) is divided into its strongly connected components by the above definition (Figure 13(b)). Its SCC-graph is depicted in Figure 13(c).

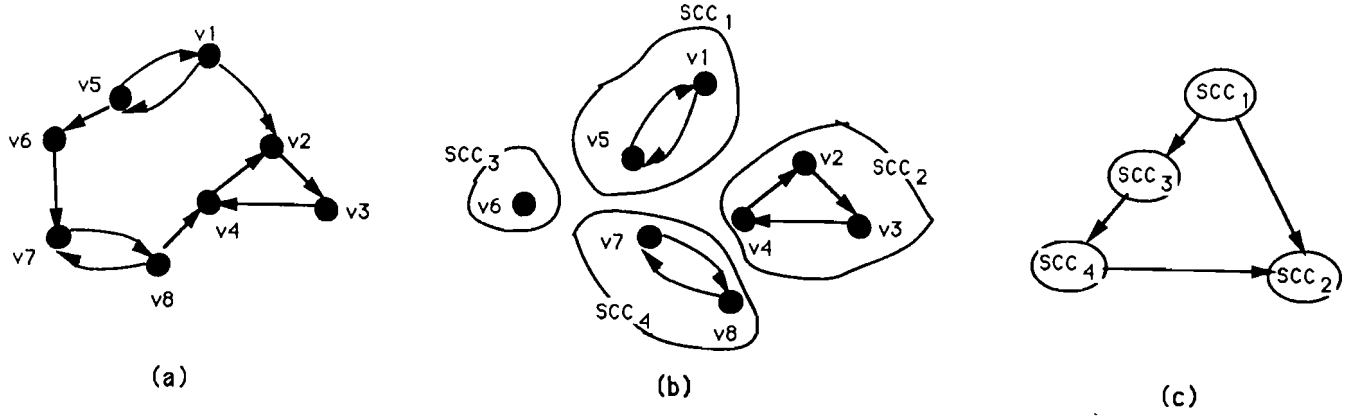



Figure 13: Directed graph $G$, SCCS($G$) and SCC-graph($G$)

We now present a simple algorithm for determining whether a template $t$ and an instance $i$ are **compatible, queue compatible** or **incompatible.** Each template can be analyzed statically when it is entered into the template library $\mathcal{L}$. For any template $\mathcal{T}$, first we compute the SCCS($G$) where $G \equiv \mathcal{T}$. Then we build two SCC-graphs($G$), one for read operations and one for writes. We perform the following analysis for each SCC-graph, first with $o$ bound to $r$ and then to $w$. We analyze each subgraph $g \in$ SCCS($G$) and label the equivalent vertex $v \in V_{scc}$ $i$ for incompatible, $c$ for compatible or $q$ for queue compatible. For each subgraph $g \in$ SCCS($G$):

- If there is any transition in $g$ of the form $\sigma =< o, \bar{m}, p, A >$, $v$ is labeled $c$.
- If there is any transition in $g$ of the form $\sigma =< o, \bar{m}, q, A >$, $v$ is labeled $q$.
- Otherwise $v$ is labeled $i$.

Now we take the transitive closure of each graph and relabel each verex $v$ as each vertex reachable from $v$ is visited. If $v$ is labeled $q$ or $c$ and some vertex reachable from $v$ is labeled $i$, $v$ is relabeled $i$. If $v$ is labeled $c$ and some vertex reachable from $v$ is labeled $q$, $v$ is relabeled $q$. Otherwise the label of $v$ isn't changed. We assume that the edges between vertices are equivalent to machines transitions that allow an operation to proceed unconditionally. If these transitions were labeled with queue or refuse, we would have to check the label of these edges while computing the transitive closure and include their values in the assignment of labels to the vertices. Given this decomposition of a template $\mathcal{T}$, we can construct a table of the submachines of $\mathcal{T}$ associated with the compatibility values computed above. Whenever an instance $i$ of $\mathcal{T}$ is executing on an object $x$ and a new template $t$ is requested for $x$, we use the predefined compatibility table to compare each operation in $t$ with the active submachine of $i$. By comparing a new template with the current and reachable submachines of active machines instead of with a static template, potential concurrency between members is improved.

We can refine our definition of a transaction group's internal protocol to include a library $\mathcal{L}' \subseteq \mathcal{L}$ of operation machines that it may execute for its members. We make no restrictions on how machines are bound to the objects used by the members. If a machine is in the group's library, any member may request that the machine be bound to any object. A possible improvement would restrict bindings so that specific objects could only be bound to specific machines. We also assume that the internal protocol includes a policy for instantiating machines. We defined templates to be compatible, queue compatible or incompatible. Possible policies are to instantiate a template $t$ for a member $m$ on an object $x$

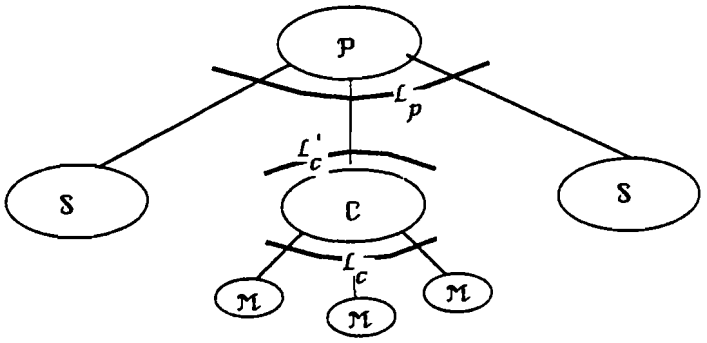



Figure 14: Mapping between sets of operation machines in TG hierarchy

1. only if $t$ is compatible with all active machines bound to $x$ or
2. if $t$ is compatible or queue compatible with all active machines bound to $x$.

Including queue compatible machines would increase the potential number of group members' that could execute concurrently although it could reduce concurrency between the members by queuing operations frequently.

## 5.4 Mapping between Machine Templates

We assumed in the previous section that a transaction group had adequate rights to instantiate and execute operation machines on behalf of its members. If the group in question is the root database, that assumption is correct. However, if the group is a member of another group and thus participates in the sharing protocol of its parent, it does not necessarily have access to an object when one of its members requires it. Part of the external protocol of a group is to provide a mapping between the machines it executes for its members and the machines that are executed on its behalf by its parent group. When a TG needs an object, it requests that a machine be instantiated by the parent group on the object. If the machine does not conflict with any machines already bound to the object, the parent group instantiates the machine returns the object to the child. In Figure 14, group $\mathcal{C}$ is a member of group $\mathcal{P}$ and has siblings labeled $\mathcal{S}$. As part of its group template, $\mathcal{C}$ provides a set of operation machines $\mathcal{L}_c$ for its members to use. Whenever a member $\mathcal{M}$ requests that a machine be instantiated by $\mathcal{C}$ on object $x$, the group must determine if any machine exists for $\mathcal{C}$ on $x$ in its parent group and whether the machine provides adequate access to object $x$. The external protocol of $\mathcal{C}$ includes a mapping between its members' machines $\mathcal{L}_c$ and the machines $\mathcal{L}'_c$ it requests from its parent $\mathcal{P}$. For example, $\mathcal{C}$ may provide multiple writer access to objects for its members. However, while the members are sharing the object, the group wants its siblings access to the object restricted to read-only. In terms of ObServer locks, the multiple writer $MW$ lock provided for group members would be mapped to a non-restrictive write lock $NW$ requested from the parent ( $MW \in \mathcal{L}_c \rightarrow NW \in \mathcal{L}'_c$ ). The parent group $\mathcal{P}$ would execute the $NW$ machine on behalf of group $\mathcal{C}$ and $\mathcal{C}$ would execute the $MW$ machines on behalf of its members.

Since our model permits the group hierarchy to change dynamically, we do not know which group will be the parent of any other group. Thus when an instance of a group is created and associated with a parent group, the database must determine whether the external protocols of the child are compatible with the internal protocols of the parent. For instance, the set of machines $\mathcal{L}'_c$ needed by the child group may not be equal to or even share any common machines with the set provided by the parent group (i.e., $\mathcal{L}'_c \cap \mathcal{L}_p = \emptyset$). In this case, another mapping between the machines in $\mathcal{L}'_c$ and $\mathcal{L}_p$ must be created.

We assume that template mapping can occur under two circumstances.

1. The external protocol of a TG template includes a mapping between the internal operation machine set $\mathcal{L}_g$ used by its members and its external operation machine set $\mathcal{L}'_g$ requested from its parent.

2. When a new TG is instantiated, the database maps the external operation machine set of the child group $\mathcal{L}'_c$ into the internal operation machine set of the parent group $\mathcal{L}_p$.

The system must be able to determine whether the proposed mappings are consistent. By consistent, we mean that the machine template instantiated for $m$ ($t_i$) does not violate the intended control of the template requested by $m$ ($t_r$). When a member requests that a template be instantiated, it expects two results. First, the operations that were permitted by $m$ in $t_r$ are permitted by $m$ in $t_i$ and second, the operations restricted for $\bar{m}$ in $t_r$ are restricted for $\bar{m}$ in $t_i$. For example, if a group requests a non-restrictive write ($NW$) machine from its parent, but the parent's operation machine set only includes restrictive write ($RW$) machines, the parent can instantiate an $RW$ machine for the member in lieu of an $NW$ machine. The $RW$ machine provides all the operations for $m$ as the $NW$ machine and restricts the same operations restricted for $\bar{m}$. However, the inverse mapping would be inconsistent since the $RW$ machine prohibits reading by $\bar{m}$ while the $NW$ machine permits $\bar{m}$ to read.

### 5.4.1 Checking the Consistency of Machine Mappings

We assume that the system must check for consistent mappings whenever a new template is added to the template library and whenever an instance of a template is instantiated and associated with a parent group. In the former instance, the TG template includes a mapping between its internal and external operation sets. This mapping must be consistent by the criteria given above. In the latter case, the system must generate a mapping between the child's external operation set and the parent's internal operation set. The algorithm for checking consistency between a requested template $t_r$ and an instantiated template $t_i$ is simple.

- for every operation $o$ that can proceed for $m$ in any state in $t_r$, $o$ must proceed for $m$ from one or more states in every submachine of $t_i$ AND
- for every operation $o$ that is queued for $\bar{m}$ from any state in $t_r$, $o$ must be queued or refused from every state in $t_i$ AND
- for every operation $o$ that is refused for $\bar{m}$ in any state in $t_r$, $o$ must be refused from every state in $t_i$.

When the system is given a set of operation machines of a group and must map them into the operation set of a parent group, it is possible that the mapping will not be one-to-one. By the above criteria, a non-restrictive read machine is consistently mapped into a non-restrictive read, a restrictive read, a non-restrictive write, or a restrictive write machine. It is not clear how to choose a machine from the relation of possible mappings. A more sophisticated algorithm would be able to identify the machine most similar to the original machine. This might include comparing the action lists of state transitions or determining whether the SCC-graphs of the machines are isomorphic. Choosing among a set of possible consistent mappings requires additional investigation.

The definition of an external protocol is refined to include a set of operation machines $\mathcal{L}'_g \subseteq \mathcal{L}$ that are used by the transaction group when requesting objects from its parent. The external protocol also includes a mapping $\mathcal{M}(\mathcal{L}_g) \rightarrow \mathcal{L}'_g$ where $\mathcal{M}$ is a consistent mapping by the criteria given above. An important external protocol not presented here is the mapping of members' write operations onto transaction group writes. It is possible that a group of cooperating applications would like to share selected objects with groups that are siblings of their parent group. Effectively, their write operations would "write through" to the next level of the hierarchy. Although not discussed here, a possible technique would be including write-through operations in the action lists of machine templates used by the group (i.e., in $\mathcal{L}'_g$).

In this section, we proposed a user-definable concurrency control mechanism, the operation machine, to enable an application builder to specify permissible orderings of operations and patterns of sharing among transaction group members. We also provided algorithms for computing compatibility between arbitrary

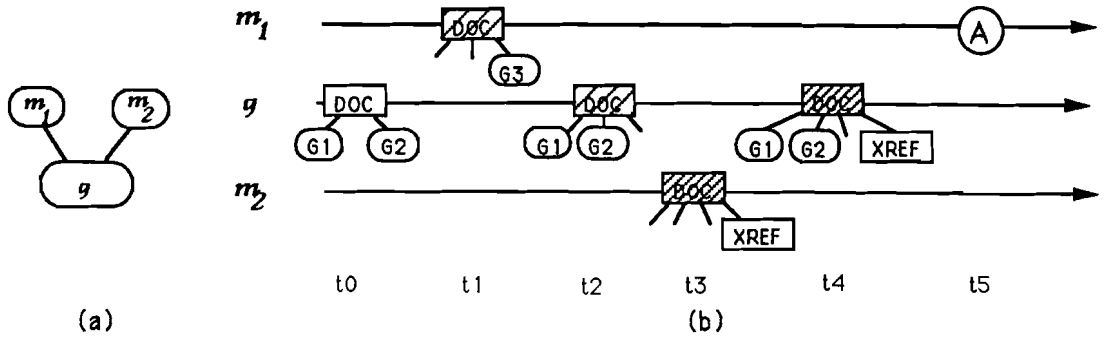



Figure 15: Abort of cooperating member $m_1$.

operation machines and for consistently mapping between sets of machines used in the transaction group hierarchy.

# 6 Recovery

A significant function of a database is to guarantee that data consistency and permanence are preserved in the face of system or transaction failures. Traditional concurrency control mechanisms rely on serializability as the criteria for guaranteeing consistency. Recovery procedures often rely on properties of the chosen concurrency control method to guarantee that operation histories are recoverable and avoid cascading aborts. In the absence of serializable executions, recovery is complicated. Our new concurrency control method permits members to share objects as long as their operation machines permit the interleavings. In this section, we extend our model of the transaction group to include mechanisms for capturing the dependencies that evolve between members sharing data and we suggest policies for recovering from member aborts. We concern ourselves only with recovery from member failure and do not address recovery from transaction group or system failure.

Since members that share objects are effectively dependent upon each others computations, member failure can leave the group's computations and data in an inconsistent state. Should a group member abort, potential inconsistencies may arise if the aborted member shared incomplete objects with its siblings or if it cached objects that were needed to maintain data consistency. Consider the group in Figure 15(a) in which members $m_1$ and $m_2$ are sharing objects. $m_1$ and $m_2$ are sharing a document **DOC** and the two graphs, **G1** and **G2**, that the document references. $m_1$ is a documentation editor and $m_2$ is a program for compiling cross-references. A time line representing the modifications to objects is in Figure 15(b). At time $t0$, the transaction group has cached the three objects. Between $t0$ and $t1$, $m_1$ reads the document, creates a new graph **G3**, and modifies the document to reference the graph. At $t2$, $m_1$ writes the **DOC** object for sharing and continues executing. Then at $t3$, $m_2$ reads the document and compiles a cross-reference object. When $m_2$ has finished, it returns the updated document and the new cross-reference object to the group. At $t5$, $m_1$ aborts and the graph which it had created and referenced in the document at $t1$ is lost. At this point, the set of objects cached by the group is incomplete since graph **G3** referenced by the document is not available. In a traditional recovery scheme, any transactions that read from an aborted transaction would be rolled back or undone completely. In a design environment, this method could sacrifice the work performed by interactive transactions and thus require people to redo work. To handle member abort in our model first requires that the dependencies between members are recognized and second requires that the group adopt a recovery policy.

## 6.1 Operation Machine Annotations

A TG executes action lists whenever a transition arc of an active machine is traversed. To capture the dependencies that evolve between members sharing data, we add default actions to the action lists of machine transitions. Before describing the new actions, we assume that the transaction group maintains two dependency graphs, called "reads from" and "over writes" graphs. We also assume that the transaction group knows the last writer of every object used by its members. The reads from ($RF$) and over writes ($OW$) graphs are directed labeled graphs defined as follows:

$$
\begin{aligned}
&RF = (V, E),\\
&OW = (V, E),\\
&V \equiv \{\text{ active members } m_i\},\\
&O \equiv \{\text{ objects shared by members }\},\\
&E = V \times O \rightarrow V
\end{aligned}
$$

The function $lw : O \rightarrow V$ takes an object identifier $x \in O$ and returns the member identifier of the last writer of $x$.

We can capture the dependencies between cooperating members by adding the following default actions to each machine template $t \in \mathcal{L}_g$. The *set*, *add* and *delete* procedures were defined in Section 5.2. For each template $t$ that is bound to object $x$,

- for each $\sigma =< r, m, p, A >\in \Delta_t$, $A = A \circ [\text{add}(rf, (m, lw(x)))]$[4]
- for each $\sigma =< w, m, p, A >\in \Delta_t$, $A = A \circ [\text{add}(ow, (m, lw(x))), \text{set}(lw(x), m)]$

Whenever member $m$ reads object $x$, a new edge $e = (m, x, \text{lw}(x))$ is added to the $RF$ graph. The edge $e$ indicates that member $m$ read object $x$ from the last writer of $x$. Whenever member $m$ writes object $x$, a new edge $e = (m, x, \text{lw}(x))$ is added to the $OW$ graph and the last writer of $x$ is set to $m$. This edge indicates that member $m$ overwrote the changes made by the last writer of object $x$.

In Figure 16, we extend the example from Figure 15 to include the dependency graphs that are created by the transaction group $g$ while cooperating members are sharing the document and graph objects. The time line also depicts the modifications made to objects by members. To compress the picture, multiple operations are performed in one time step. At $t0$, the group has the **DOC**, **G1** and **G2** objects cached for member use. At time $t1$, $m_1$ modifies **DOC**, $m_2$ modifies **G1** and $m_3$ modifies **G2** and at time $t2$, each member updates the objects in the group cache. We assume that cooperating members are notified of the updates (this can be realized by using *notify(m, update)* actions). Then each member reads the new versions of the objects at $t3$. This forces the group to update its $RF$ graph. Notice that edges in the graph are labeled with the identifier of the object that was shared by two members. Cycles may exist in this graph if two or more members read objects from each other. At $t4$, $m_4$ updates the document and creates a new graph **G3**. This modification adds an edge to the $OW$ graph and changes the last writer of **DOC** to $m_4$. In the last time step, $m_2$ and $m_3$ reread the document and two more edges are added to the $RF$ graph. At time $t4$, the set of objects cached by the group is incomplete since member $m_4$ has not yet written graph **G3** for sharing.

## 6.2 Synchronization Points

Because our model cannot decipher internal object formats, it cannot recognize the explicit dependencies that exist between objects. We assume however that a member application can recognize the dependencies between objects it accesses. Although the TG cannot determine when the objects it has cached are consistent,

[4] The $\circ$ symbol denotes list concatenation.

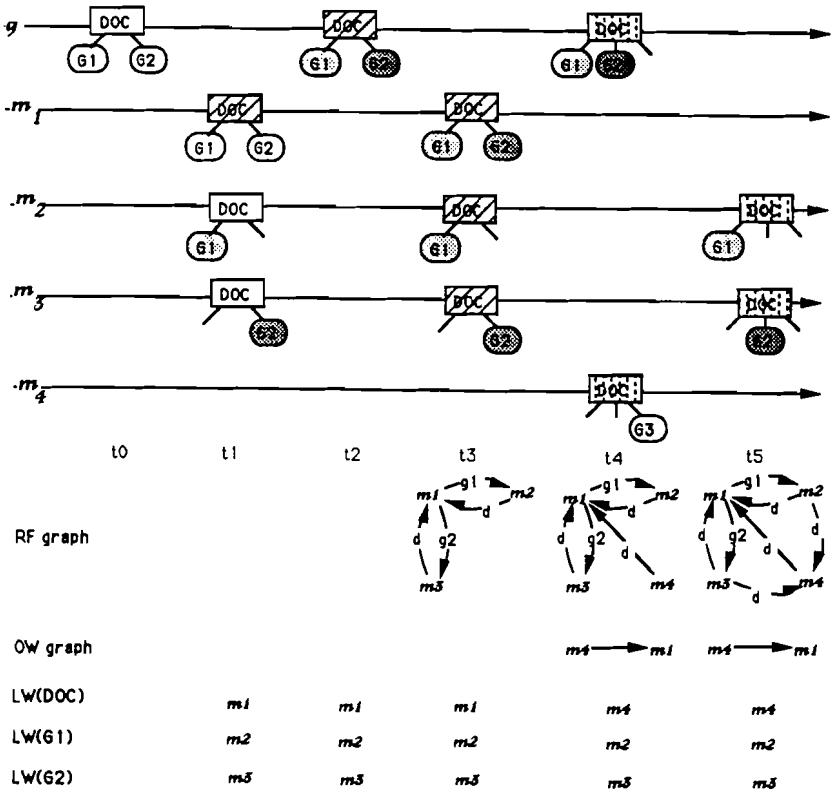



Figure 16: Dependency graphs built by transaction group $g$

it can request that members "synchronize" their work by returning or updating objects that are necessary for a consistent view. The TG captures dependencies between members sharing objects in the $RW$ and $OW$ graphs and can report to a member $m$ the set of objects that other members have read from $m$, the set $m$ has read from other members, the set that $m$ has overwritten or the set that have been over written by other members. Given this information, $m$ can update or return the set of objects necessary to make the reported set of objects complete. This places the onus on a member to have access to or be able to determine a consistent view of objects. We choose to compute sets of objects so that each member can return the minimum set of objects necessary to ensure consistency of shared work. For example, if member $m$ shares a set of objects $O'$ with member $m'$ and a disjoint set of objects $O''$ with member $m''$ (i.e., $O' \cap O'' = \emptyset$) and then $m'$ requests that any objects it used be updated and made consistent for the group, member $m$ is notified to update only those objects in $O'$.

We define two new messages that a TG and a member application can use to request and acknowledge synchronization.

| | |
|---|---|
| $sync(m, O)$ | The TG requests that member $m$ synchronize with respect to the set of objects $O$. |
| $ack(m, O, O')$ | Member $m$ acknowledges a synchronization request and returns the set of objects $O'$ dependent upon objects in set $O$ ($O \subseteq O'$). |

To illustrate the use of a synchronization request, consider member $m_4$ at time $t4$ in Figure 16. At this point in the timeline, $g$ does not have a consistent view of the document since the graph **G3** is not accessible. By requesting that $m_4$ synchronize with respect to **DOC**, the TG places the responsibility on the application to recognize any dependencies between **DOC** and **G3** and to report them. The request *sync(m,* {**DOC**}*)* would be acknowledged with *ack(m,* {**DOC**}, {**G3**}*)*. Each *ack* operation received by the TG generates a *synchronization point* in the operation history of the member.

Since the purpose of an *ack* operation is to notify the group that its view with respect to a set of objects is consistent according to member $m$, we remove any dependencies that relied upon member $m$'s modifications to the set of objects. This is illustrated in Figure 17. This time line is an extension of the

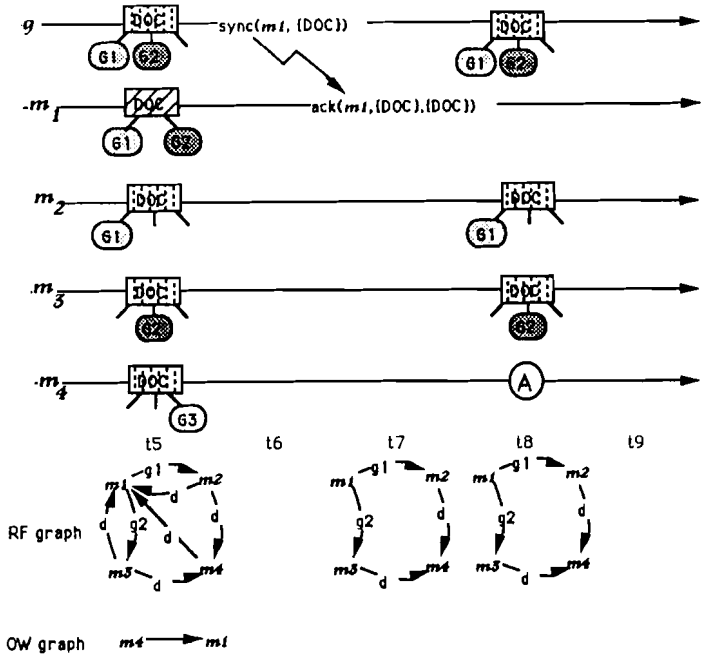



Figure 17: Synchronization point in operation history

example in Figure 16 beginning at time $t5$. At time $t6$, the group requests that member $m_1$ synchronize with respect to the object set {**DOC**}. Since $m_1$ only modified **DOC**, it acknowledges the request by returning just the **DOC** object. The acknowledgement indicates to the group that **DOC** is complete with respect to $m_1$ and that any members previously dependent on $m_1$'s changes to **DOC** are no longer dependent upon that member. This permits the group to remove dependencies from the $RF$ graph. In addition to reducing intermember dependencies, synchronization points are intended to limit the extent of rollbacks during recovery operations.

## 6.3 Member Commit

As previously stated, member commit connotes successful completion of a computation and termination of the member application. Since we assume that member applications are interactive, requiring a member to redo previously executed operations is undesireable. Thus when a member commits, the group should be able to guarantee that the member's work will be both "consistent" with respect to other members' work and "permanent" in that it will not be undone at a later point in time. A possible approach is to require the member to wait for all members that it read from, and thus was dependent upon, to commit. This limits the autonomous nature of interactive applications and reduces a group of independent designers to a tightly interdependent group. We prefer to request that cooperating members synchronize their work so that the committing member is not dependent upon any other members' cached objects. The TG relies on the $RF$ and $OW$ graphs to determine the set of members with which the committing member shared objects.

A member commit operation triggers the transaction group to issue synchronization requests to every member upon which the committing member was dependent. For each member $m_i$ from which the committing member $m$ read, the TG adds $m_i$ to a set $M$ and constructs a set of the objects $O_{m_i}$ that were read by $m$. The TG then requests synchronization on the object sets by each member in $M$. The operation sequence for constructing the sets follows.

1. for each $m_i \in V, o \in O$ such that $e = (m, o, m_i) \in E$,

add $m_i$ to set $M$,
add $o$ to set $O_{m_i}$.

2. for each $m_i \in M$, sync$(m_i, O_{m_i})$.

The commit for $m$ remains pending until all members in $M$ acknowledge the synchronization request. Each acknowledgement triggers the transaction group to remove dependencies from the $RF$ graph. The acknowledge operation $ack(m_i, O_{m_i}, O'_{m_i})$ results in the following sequence of operations.

1. for each $o \in O_{m_i}$ or $\in O'_{m_i}$,
   if $lw(o) \equiv m_i$, update $o$.
   for each $m_j$ such that $e = (m_j, o, m_i) \in E$,
   delete(RF, $(m_j, o, m_i)$)

When every edge in the $RF$ graph of the form $e = (m, o, m_i)$ has been deleted, the pending commit for member $m$ is completed.

The acknowledgements of synchronization requests made by the TG comprise an agreement on the part of the members that the view maintained by the TG is consistent. In an aberrant case, a member could respond to a *sync* request with an empty *ack* response and the TG would not be able to determine that the member was being deviant. However, at the object server level where there is no access to a high level type system that explicitly defines interobject dependencies, the system can only provide mechanisms for supporting the cooperation of member applications. It cannot guarantee that the member applications will abide by the protocols.

## 6.4 Member Abort

In our model, member abort connotes the failure of a member's computation due either to internal errors (e.g., program bug) or to external causes such as network partitions or system crashes. We assume that the TG can identify a member abort either by receiving an explicit abort operation from a member or by detecting that the member is not accessible. Regardless of the cause, the TG can expect that the member will not complete its computation or respond to any messages. Since the model relies upon members to respond to synchronization requests when group members commit, a member abort can result in the indefinite delay of another member's commit. Consider the example in Figure 17 beginning at time $t8$ when member $m_4$ aborts. At this point, if member $m_2$ or $m_3$ attempted to commit, the transaction group would not be able to notify member $m_4$ to synchronize and would never delete the dependencies that exist in the $RF$ graph. The member commits would be delayed indefinitely. Possible policies for handling member abort include

- warning dependent members of possible inconsistencies and removing edges from the dependency graphs,
- warning dependent members and waiting for an alternate member to compensate for the aborted member or
- undoing dependent members' operations to a point in the group's history when no members were dependent upon the aborted member.

We examine the effects of each of these policies.

If the TG adheres to the first policy, whenever a member aborted, any members dependent upon the aborted member's work would be warned of possible inconsistencies. The group would then remove any dependencies from the $RW$ and $OW$ graphs. This would permit members dependent upon an aborted member to commit when required. However, this does not guarantee that the transaction group object

cache is consistent with respect to all members' changes. The group could keep track of aborted members updates and require human intervention to check for consistency before any objects were written through to the parent group.

The second policy would require that an alternate member compensate for the aborted member. The dependency graphs would remain unchanged until the compensating member was chosen. Its identifier would be substituted for the aborted member's identifier in the graphs. The *OW* graph can be used to choose an alternate member. It is possible that any application that overwrote an aborted member would know how to redo (or reconstruct) the necessary changes. The TG could use the *OW* graph to notify possible substitutes that a member had aborted. Although the group cannot ensure that some compensating member will volunteer, it can use the dependency graphs as clues to closely related members.

The last policy takes a traditional tack to recovery by requiring members dependent on an aborted member to rollback or undo. In the example of Figure 17, members $m_2$ and $m_3$ would be rolled back to the last synchronization point in their histories when neither was dependent upon $m_4$'s updates. Since neither member had previously synchronized, this would require rolling back to time $t0$. However, $m_1$ read from $m_2$ and $m_3$ so it would also be rolled back. To avoid cascading rollbacks and undoing committed transactions, we must limit rollbacks to the latest synchronization point in the group's history. Since $m_1$ synchronized at $t7$, the rollback of $m_1, m_2$ and $m_3$ due to the abort of $m_4$ would cease at that point. It is possible that this approach does not restore consistency if the latest synchronization point is later than the earliest point in the history when no member depended upon the aborted member. In this case, the problem reduces to the solution of the first policy in which members are warned of inconsistencies but are permitted to proceed. By increasing the frequency of synchronization points, the dependency graphs remain sparsely connected and thus rollbacks have a less detrimental effect on interactive applications. Obviously, the overhead of synchronizing frequently is high. Fewer synchronization points would result in the undesirable effects of rollbacks that undo long periods of work or data inconsistency in the group's object cache because needed objects were lost.

The choice of a recovery policy by a transaction group depends primarily upon the resilience of the application programs. If one application program can easily compensate for another, the group can guarantee that all intermember dependencies are observed when members commit or abort. We do not unilaterally choose a recovery policy since the the functionality of the cooperating applications dictates how effective or desirable a particular policy may be.

# 7 Future Research

In this section, we present some problems not addressed in this paper such as how to efficiently cache objects in transaction groups, how to handle recovery from TG failures and the possible application of our machine compatibility algorithm to operation machines defined on complex operations.

## 7.1 Distributed Transaction Groups

We have presented a logical model that captures the hierarchical organization of design transactions. We have assumed that object caches and operation machines exist at each level in the transaction group hierarchy and that groups instantiate machines and move objects through the hierarchy as needed. The abstract model does make any assumptions about the physical location(s) of transaction groups or their members. Since the primary platforms for design applications are networks of engineering workstations, an implementation of transaction groups must consider that applications and groups are distributed across multiple sites. In Figure 18, note that design transactions $T_1$ and $T_3$ are executing on the same machine but are members of different groups. Also note that the groups "TGengine" and "TGengine_design" execute at different

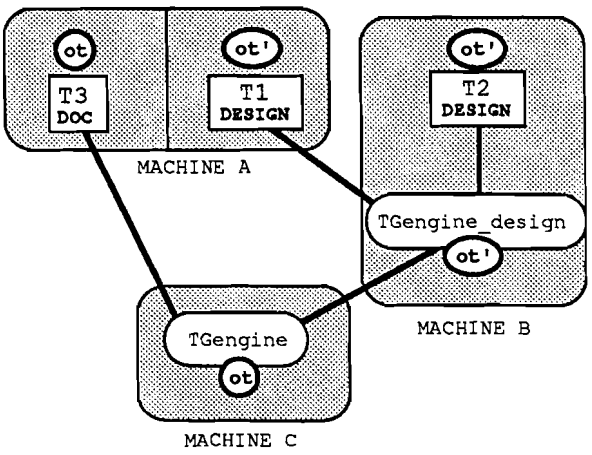



Figure 18: Machine boundaries between transaction groups and design applications

sites. Issues for consideration include the efficient movement of objects between distributed object caches and handling distributed commit. Certainly, research from distributed databases can provide many useful models and algorithms applicable to this problem.

## 7.2 Transaction Group Recovery

In Section 6, we were concerned primarily with mechanisms to handle recovery due to member failure. The underlying assumption was that the failing member was a design application, not another transaction group. We did not address the possibility that the transaction group, like any database, could fail itself. This possibility raises two issues. First, the transaction group must have internal recovery procedures for restoring consistency to its own caches and logs and for possibly recontacting its members so they may continue processing. Second, the failed transaction group is a member of another transaction group which can (probably) identify that one of its members has failed. The parent group has its own protocol for handling recovery due to member failure. Ideally, the parent group should not have to know that the failing member was an application or another group. However, in an aberrant case, the child group might not have any recovery procedure. If the parent's recovery policy was to wait for a compensating member as a substitute for the failed member, the parent group could wait indefinitely. This example demonstrates that a group's recovery protocol may have undesirable non-local effects in the hierarchy. Additional research on recovery is needed in this area.

## 7.3 Machine Compatibility for Complex Operations

The simplifications of Skarra's operation machine model enabled us to simply define and easily compute machine compatibility . An interesting use of this technique would be applying it to machines defined for abstract data types that use complex operations. Skarra's model includes additional properties of machines such as local variables (to maintain internal state) and complex predicates for determining operation conflict. It is not clear whether our algorithm would be able to compute the compatibility of two arbitrary machines defined on complex operations instead of read and write. This problem is worth investigating as a solution might simplify or eliminate having to define operation machines between different types.

# 8 Conclusion

From our work on designing and developing ObServer, we have recognized the need for an extensible transaction model that can support possibly non-serializable, hierarchically-organized design applications. We have presented a new mechanism, the transaction group, for specifying the logical groupings of design transactions and a model for building a transaction group hierarchy that reflects the organizational structure of a design environment. Our model allows the TG hierarchy to be created dynamically and to be reorganized as design groups change. The model encapsulates group interactions so that heterogeneous views of correctness can coexist in the same object server. We require a transaction group to adhere to internal and external protocols. These protocols include mechanisms for concurrency control and recovery of members. In lieu of predefined locks, we provide user-definable operation machines that capture the semantics of existing ObServer locks as well as the permissible interleavings of cooperating members' operations. Operation machines provide a formal alternative to strictly serializable executions. The transaction group supports recovery procedures that capture the dependencies that evolve between members sharing data and uses these dependencies to inform members of possible inconsistencies when members abort. We also provide a framework for consistently mapping between the internal protocols of the group and the external interaction with a parent group. This mapping guarantees that each level in the hierarchy adheres to its own view of correctness without compromising the integrity of other groups in the hierarchy.

An implementation of our new ObServer model and additional investigation is necessary to determine whether our primitives for concurrency control and recovery provided at the object server level can effectively support the requirements of object-oriented database programming languages, hypermedia systems and interactive programming environments.

# 9 Acknowledgements

I would like to thank my advisor Stan Zdonik for providing motivation and encouragement to think about the "hard" problems. Many thanks go to Andrea Skarra for solving the hard problems and for interesting conversations on her thesis. Thanks also go to Marian Nodine for constructive criticism on this paper and lots of moral support. And deep gratitude goes to dāv Lion and Alan Potter for making ObServer live and breathe and for being such fun to work with.

# References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

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

[FZ89] Mary Fernandez and Stanley Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Third International Workshop On Persistent Object Systems*, January 1989.

[HZ87] Mark Hornick and Stanley Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5:70–95, January 1987.

[K+84] W. Kim et al. A transaction mechanism for engineering design databases. In *International Conference on VLDBs, Singapore*, 1984.

[K+85] P. Klahold et al. A transaction model supporting complex applications in integrated information systems. *ACM SIGMOD*, 1985.

[KKB87] H. Korth, W. Kim, and F. Bancihon. On long-duration CAD transactions. *Information Systems*, 13, 1987.

[LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Databases for Engineering Applications*, pages 115–121. ACM, May 1983.

[Lyn83] Nancy A. Lynch. Multilevel atomicty - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4), December 1983.

[MHL+89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Technical Report RJ 6649, IBM Almaden Research Center, January 1989.

[R+88] S. Rehm et al. Support for design processes in a structurally object-oriented database system. In *Second International Workshop on Object-Oriented Database Systems*, pages 80–96. ACM, 1988.

[Ska88] Andrea Skarra. Concurrency control for cooperating transactions in an object-oriented database. Technical report, Brown University, September 1988.

[Ska89] Andrea Skarra, 1989. Personal communication and forthcoming Ph.D. thesis.

[Wei89] William Weihl. The impact of recovery on concurrency control. *ACM PODS*, pages 259–269, 1989.