Synchronizing Shared Abstract Types

(Revised Issue)

Peter M. Schwarz and Alfred Z. Spector

18 November 1983

Abstract

This paper discusses the synchronization issues that arise when transaction facilities are extended for use with shared abstract data types. A formalism for specifying the concurrency properties of such types is developed, based on dependency relations that are defined in terms of an abstract type's operations. The formalism requires that the specification of an abstract type state whether or not cycles involving these relations should be allowed to form. Directories and two types of queues are specified using the technique, and the degree to which concurrency is restricted by type-specific properties is exemplified. The paper also discusses how the specifications of types interact to determine the behavior of transactions. A locking technique is described that permits implementations to make use of type-specific information to approach the limits of concurrency.

Technical Report CMU-CS-83-163, Revision of CMU-CS-82-128

This research was sponsored by: the USAF Rome Air Development Center under contract F30602-81-C-0297; the US Naval Ocean Systems Center under contract number N66001-81-C-0484 N65; and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Table of Contents

1 Introduction

Transactions facilities, as provided in many database systems, permit the definition of *transactions* containing operations that read and write the database and that interact with the external world. The transaction facility of the database system guarantees that each invocation of a transaction will execute at most once (i.e., either commit or abort) and will be isolated from the deleterious effects of all concurrently executing transactions. To make these guarantees, the transaction facility manages transaction synchronization, recovery, and, if necessary, inter-site coordination. Many papers have been written about transactions in the context of both distributed and non-distributed databases [Bernstein 81, Eswaran 76, Gray 80, Lampson 81, Lindsay 79].

There are a number of ways in which transaction facilities could be extended to simplify the construction of many types of reliable distributed programs. Extensions that allow a wider variety of operations to be included in a transaction would facilitate manipulation of shared objects other than a database. Extensions that permit transaction nesting would facilitate more flexible program organizations, as would extensions allowing some form of inter-transaction communication of uncommitted data. Although the synchronization, recovery, and inter-site coordination mechanisms needed to support database transaction facilities are reasonably well understood, these mechanisms require substantial modification to support such extensions. For example, they must be made compatible with the abstract data type model and with general implementation techniques such as dynamic storage allocation.

Lomet [Lomet 77] considered some of the problems encountered in developing general-purpose transaction facilities, but more recently, much of the research in this area has been done at MIT. Moss and Reed have discussed nested transactions and other related systems issues [Moss 81, Reed 78]. As part of the Argus project, extensions to CLU have been proposed that incorporate primitives for supporting transactions [Liskov 82a, Liskov 82b]. Additionally, Weihl has considered transactions that contain calls on shared abstract types such as sets and message queues, and has discussed their implementation [Weihl 83a, Weihl 83b]. Transactions will also be available in the Clouds distributed operating system [Allchin 83].

This paper focuses on one important issue that arises when extending transaction facilities: the synchronization of operations on shared abstract data types such as directories, stacks, and queues. After a presentation of background material in the following section, Section 3 introduces some tools and notation for specifying shared abstract types. Section 4 describes three particular data types and uses the tools to specify how operations on these types can interact under conditions of concurrent access by multiple transactions. The specifications that are developed make explicit use of type-specific properties, and it is shown how this approach permits greater concurrency than standard techniques that do not use such information. Section

5 discusses how the specifications of individual types interact to determine global properties of groups of transactions. Section 6 proposes an extensible approach to locking that can be used for synchronization in implementations intended to meet these specifications. Finally, Section 7 summarizes the major points of this paper and concludes with a brief discussion of other considerations in the implementation of user-defined, shared abstract data types.

2 Background

Transactions aid in maintaining arbitrary application-dependent *consistency constraints* on stored data. The constraints must be maintained despite failures and without unnecessarily restricting the concurrent processing of application requests.

In the database literature, transactions are defined as arbitrary collections of database operations bracketed by two markers: *BeginTransaction* and *EndTransaction.* A transaction that completes successfully *commits*; an incomplete transaction can terminate unsuccessfully at any time by *aborting.* Transactions have the following special properties:

1. Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity.*

2. If a transaction completes successfully, the effects of its operations will never subsequently be lost. This property is usually called *permanence.*

3. If a transaction aborts, no other transactions will be forced to abort as a consequence. *Cascading aborts* are not permitted.

4. If several transactions execute concurrently, they affect the database as if they were executed serially in some order. This property is usually called *serializability.*

Transactions lessen the burden on application programmers by simplifying the treatment of failures and concurrency. Failure atomicity makes certain that when a transaction is interrupted by a failure, its partial results are undone. Programmers are therefore free to violate consistency constraints temporarily during the execution of a transaction. Serializability ensures that other concurrently executing transactions cannot observe these inconsistencies. Permanence and prevention of cascading aborts limit the amount of effort required to recover from a failure. Transaction models that do not prohibit cascading aborts are possible, but we do not consider them.

Our model for using transactions in distributed systems differs from this traditional model in several ways. The most important difference is that we incorporate the concept of an *abstract data type.* That is, information is stored in typed *objects* and manipulated only by *operations* that are specific to a particular object type. The users of a type are given a *specification* that describes the effect of each operation on the stored data, and new abstract types can be implemented using existing ones. The details of how objects are represented and how the operations are carried out are known only to a type's implementor. Abstract data types grew out of the class construct in Simula [Dahl 72], and are supported in many other programming languages including CLU [Liskov 77], Alphard [Wulf 76], and Ada [Dept. of Defense 82], as well as in operating systems, e.g. Hydra [Wulf 74]. In our system model, transactions are composed of operations on objects that are instances of abstract types. Of particular interest are those objects that are not local to a single transaction. These are instances of *shared abstract types*.

We assume that the facilities for implementing shared abstract types and for coordinating the execution of transactions that operate on them are provided by a basic system layer that executes at each node of the system. This *transaction kernel* exports primitives for synchronization, recovery, deadlock management, and inter-site communication. In some ways, a transaction kernel is similar to the RSS of System R [Gray 81]. A transaction kernel, however, is intended to run on a bare machine and must supply primitives useful for implementing arbitrary data types, whereas the RSS has the assistance of an underlying operating system and only provides specialized primitives tailored for manipulating a database.

Another difference between our system model and the traditional transaction model is that we do not necessarily require that transactions appear to execute serially. Serializability ensures that if transactions work correctly in the absence of concurrency, any interleaving of their operations that is allowed by the system will not affect their correctness. But sometimes, serializability is too strong a property, and requiring it restricts concurrency unnecessarily. For example, it is usually unnecessary for two letters mailed together and addressed identically to appear in their recipient's mailbox together. However, serializability is violated if the letters do not arrive contiguously, because there is no longer the appearance that the sender has executed without interference from other senders. Thus, it may be desirable for some shared abstract types to allow limited non-serializable execution of transactions. This idea has also been investigated by Garcia-Molina [Garcia-Molina 83] and Sha et al. [Sha 83].

Serializability guarantees that an ordering can be defined on a group of transactions. If the transactions share some common objects, serializability requires that these objects be visited in the same order by all the transactions in the group. In the next section, a more general ordering property of transactions is defined, of which serializability is a special case. We will show that it is possible to prove that transactions work correctly in the presence of concurrency, even if they do not appear to execute serially.

In order to maintain the special properties of transactions in our model, the operations on shared abstract types that compose them must meet certain requirements. To guarantee the failure atomicity of transactions, it must be possible to undo any operation upon transaction abort. Therefore, an *undo operation* must be provided for each operation on a shared abstract type. Recovery is not the main concern of this paper, and we will be considering undo operations only as they pertain to synchronization issues. Further discussion of recovery issues can be found in a related paper [Schwarz 83].

Operations on shared abstract types must also meet three synchronization requirements:

1. Operations must be protected from anomalies that could be caused by other concurrently executing operations on the same object. Freedom from these concurrency anomalies ensures that an invocation of an operation on a shared object is not affected by other concurrent operation invocations. This is the same property that monitors provide [Hoare 74].

2. To preclude the possibility of cascading aborts, operations on shared objects must not be able to observe information that might change if an uncommitted transaction were to abort. This may necessitate delaying the execution of operations on behalf of some transactions until other transactions complete, either successfully or unsuccessfully.

3. When a group of transactions invokes operations on shared objects, the operations may only be interleaved in ways that preserve serializability or some weaker ordering property of the group of transactions. The synchronization needed to control interleaving cannot be localized to individual shared objects, but rather requires cooperation among all the objects shared by the transactions.

Traditional methods for synchronizing access to an instance of a shared abstract type are designed solely to ensure the first goal: correctness of individual operations on an object. This paper is concerned with the second and third goals. We examine the problem of specifying the synchronization needed to achieve them, as well as the support facilities that the transaction kernel must provide to implementors of shared abstract types.

3 Dependencies: A Tool for Reasoning About Concurrent Transactions

This section introduces a theory that can be used to reason about the behavior of concurrent transactions. It allows the standard definition of serializability to be recast in terms of shared abstract types, and provides a convenient way of expressing other ordering properties. The theory is also useful in understanding cascading aborts.

3.1 Schedules

Schedules [Eswaran 76, Gray 75] can be used to model the behavior of a group of concurrent transactions. Informally, a schedule is a sequence of <transaction, operation> pairs that represents the order in which the component operations of concurrent transactions are interleaved. Schedules are also known as *histories* [Papadimitriou 77] and *logs* [Bernstein 79]. In some of the traditional database literature, the operations in schedules are assumed to be arbitrary; no semantic knowledge about them is available [Eswaran 76]. In this case, a schedule is merely an ordered list of transactions and the objects they touch:

```
T1: O1
T2: O1
T2: O2
T1: O2
```

In other work, operations are characterized as Read(R) or Write(W) [Gray 75], in which case the schedule includes that semantic information:

```
T1: R(O1)
T2: R(O1)
T2: W(O2)
T1: R(O2)
```

To analyze transactions that contain operations on specific shared abstract types, we will consider schedules in which these operations are characterized explicitly. For example, a schedule may contain operations to enter an element on a queue or to insert an entry into a directory. We call these *abstract schedules*, because they describe the order in which operations affect objects, regardless of any reordering that might be done by their implementation.[1] Given the initial state of a set of objects, an abstract schedule of operations on these objects, and specifications for the operations in the schedule, the result of each operation and the final state of the objects can be deduced. For instance, consider the following abstract schedule, which is composed of operations on Q, a shared object of type FIFO Queue. The operations **QEnter** and **QRemove** respectively append an element to the tail of a FIFO Queue and remove one from it's head. Assume Q to be empty initially.

```
T1: QEnter(Q, X)
T2: QEnter(Q, Y)
T3: QRemove(Q)
```

From this abstract schedule and the initial contents of the Queue, one can deduce the state of Q at any point in the schedule. Thus one may conclude that the **QRemove** operation returns X, and that only Y remains on the Queue at the end of the schedule.

3.2 Dependencies and Consistency

By examining an abstract schedule, it is possible to determine what *dependencies* exist among the transactions in the schedule. The notation D: $T_i{:}X \rightarrow_O T_j{:}Y$ will be used to represent the dependency D formed when transaction T_i performs operation X and transaction T_j subsequently performs operation Y on some common object O. The object, transaction, or dependency identifiers may be omitted when they are unimportant. The set of ordered pairs $\{(T_i, T_j)\}$ for which there exist X, Y and O such that D: $T_i{:}X \rightarrow_O T_j{:}Y$ forms a relation, denoted $<_D$. If $T_i <_D T_j$, T_i *precedes* T_j and T_j *depends* on T_i, under the dependency D.

[1]In Section 4.4 we will define a second kind of schedule, the *invocation schedule*, which reflects the concurrency of specific implementations.

Examples of dependencies and their corresponding relations can be drawn from traditional database systems. For instance, consider a system in which no semantic knowledge, either about entire transactions or about their component operations, is available to the concurrency control mechanism. The only requirement is that each individual transaction be correct in itself: it must transform a consistent initial state of the database to a consistent final state. Under these conditions, only serializable abstract schedules can be guaranteed to preserve the correctness of individual transactions.

Since all operations are indistinguishable, only one possible dependency D can be defined: $T_1 <_D T_2$ if T_1 performs any operation on an object later operated on by T_2. Now, consider $<^*_D$, the transitive closure of $<_D$. A schedule is *orderable* with respect to $\{<_D\}$ iff $<^*_D$ is a partial order. In other words, there are no cycles of the form $T_1 <_D T_2 <_D \ldots <_D T_n <_D T_1$. In general, a schedule is orderable with respect to S, where S is a set of dependency relations, iff each of the relations in S have a transitive closure that is a partial order. The relations in S are referred to as *proscribed* relations, and we will use orderability with respect to a set of proscribed dependency relations to describe ordering properties of groups of transactions. Abstract schedules that are orderable with respect to a specified set of proscribed relations will be called *consistent* abstract schedules.

It can be shown that orderability with respect to $\{<_D\}$ is equivalent to serializability [Eswaran 76]. Given a schedule orderable with respect to $\{<_D\}$, a transaction T, and the set O of objects to which T refers, every other transaction that refers to an object in O can unambiguously be said either to precede T or to follow T. Thus T depends on a well-defined set of transactions that precede it, and a well-defined set of transactions depend on T. Each transaction sees the consistent database state left by those transactions that precede it, and (by assumption) leaves a consistent state for those that follow. The set of schedules for which $<^*_D$ is a partial order constitutes the set of consistent abstract schedules for a system that employs no semantic knowledge.

The scheme described above prevents cycles in the most general possible dependency relation, hence it maximally restricts concurrency. By considering the semantics of operations on objects, it is possible to identify some dependency relations for which cycles may be allowed to form. For example, consider a database with a Read/Write concurrency control. Such systems recognize two types of operations on objects: Read(R) and Write(W). Thus there are 4 possible dependencies between a pair of transactions that access a common object:

- D_1: $T_i{:}R \rightarrow_O T_j{:}R$. T_i reads an object subsequently read by T_j.
- D_2: $T_i{:}R \rightarrow_O T_j{:}W$. T_i reads an object subsequently modified by T_j.
- D_3: $T_i{:}W \rightarrow_O T_j{:}R$. T_i modifies an object subsequently read by T_j.

- D_4: $T_i{:}W \rightarrow_O T_j{:}W$. T_i modifies an object subsequently modified by T_j.

The earlier scheme, by not distinguishing between these dependencies, prevents cycles from forming in the dependency relation $<_D$, which is the union of all four individual relations. By contrast, Read/Write concurrency controls take into account the fact that $R \rightarrow R$ dependencies cannot influence system behavior. That is, given a pair of transactions, T_1 and T_2, and an abstract schedule in which both T_1 and T_2 perform a Read on a shared object, the semantics of Read operations ensure that neither T_1, T_2 nor any other transaction in the schedule can determine whether $T_1 <_{D_1} T_2$ or $T_2 <_{D_1} T_1$. Since these dependencies cannot be observed, they cannot compromise serializability, nor can they affect the outcome of transactions. We call dependencies meeting this criterion *insignificant*. Korth has also noted that when operations are commutative, their ordering does not affect serializability [Korth 83].

For the Read/Write case, the necessary condition for serializability can be restated as follows in terms of dependency relations: a schedule is serializable if it is orderable with respect to $\{<_{D_2 \cup D_3 \cup D_4}\}$ [Gray 75]. By allowing multiple readers, Read/Write schemes permit the formation of cycles in the $<_{D_1}$ dependency relation, and in relations that include $<_{D_1}$, while preventing cycles in the relation that is the union of $<_{D_2}$, $<_{D_3}$ and $<_{D_4}$. For example, consider the following schedules, which have identical effects on the system state:

```
T1: R(O1)              T2: R(O1)
T2: R(O1)              T1: R(O1)
T1: W(O1)              T1: W(O1)
```

In the first schedule, $T_1 <_{D_1} T_2$ and $T_2 <_{D_2} T_1$. Hence, there is a cycle in the relation $<_{D_1 \cup D_2}$, although $<_{D_2 \cup D_3 \cup D_4}$ is cycle-free. In the second schedule, the first two steps are reversed and neither cycle is present.

On the other hand, the following two schedules are not necessarily identical in effect:

```
T1: R(O1)              T2: W(O1)
T2: W(O1)              T1: R(O1)
T1: W(O1)              T1: W(O1)
```

In this case, the first schedule is not serializable because $T_1 <_{D_2} T_2$ and $T_2 <_{D_4} T_1$, thus forming a cycle in the relation $<_{D_2 \cup D_4}$, which is a sub-relation of $<_{D_2 \cup D_3 \cup D_4}$. T_1 observes O_1 before it is written by T_2, but the final state of O_1 reflects the Write of T_1 rather than T_2, implying that T_1 ran after T_2. The second schedule has no cycle and is serializable.

In summary, orderability with respect to a set of proscribed dependency relations provides a precise way to characterize consistent schedules. For a concurrency control that enforces serializability with no semantic knowledge at all about operations, the set of proscribed relations must contain $<_D$, which is equivalent to the union of every possible dependency relation. For a Read/Write database scheme, the set contains the $<_{R \rightarrow W \,\cup\, W \rightarrow R \,\cup\, W \rightarrow W}$ relation. When type-specific semantics are considered, type-specific dependency relations can be defined for each type. In Section 4, dependencies are used to define *interleaving specifications* for various abstract types. These specifications provide the information needed to determine how an individual type can contribute toward maintaining a global ordering property such as serializability. If a specification guarantees orderability with respect to the union of all significant dependency relations for a given type, then it is strong enough to permit serializability. In general, however, more concurrency can be obtained when only weaker ordering properties are guaranteed. The way in which the interleaving specifications of multiple types interact to preserve global ordering properties is discussed in Section 5.

3.3 Dependencies and Cascading Aborts

Dependencies are also useful in understanding cascading aborts. A cascading abort is possible when a dependency forms between two transactions, the first of which is uncommitted. An abort by this uncommitted transaction may cascade to those that depend on it. Whether or not a cascade actually must occur depends on the exact type of dependency involved, and the properties of the object being acted upon. For example, consider the four general dependency relations that arise in Read/Write database systems. R $\rightarrow$ R dependencies are insignificant, and can never cause cascading aborts. This is analogous to the role of these dependencies in determining orderability. Likewise, R $\rightarrow$ W and W $\rightarrow$ W dependencies need not cause cascading aborts, because in both cases the outcome of the second transaction does not depend on data modified by the first[2]. By contrast, W $\rightarrow$ R dependencies represent a transfer of information between the two transactions. In the absence of any additional semantic information, it must be assumed that an abort of the first transaction will affect the outcome of the second, which must therefore also be aborted.

Once the dependencies that could lead to cascading aborts have been identified, their formation must be controlled. Stated in terms of abstract schedules: starting from the first of the two operations that form the dependency there must be no overlapping of the two transactions in the schedule, with the prior transaction in the dependency relation completing first. Such schedules will be called *cascade-free.* Note that some consistent schedules may not be cascade-free, and vice-versa.

4 Specification of Shared Abstract Types

This section focuses on the typed operations that make up transactions and discusses how to specify their local synchronization properties. The traditional specification of an abstract type describes the behavior of the type's operations in terms of preconditions, postconditions, and an invariant. This specification must be augmented in several ways to complete the description of a shared abstract type in our model. In the first place, the undo operation corresponding to each regular operation must be specified in terms of

[2]It may be necessary to control the formation of these dependencies anyway, if an insufficiently flexible recovery strategy is used.

preconditions, postconditions and the invariant. Specification of the undo operations themselves is not considered further in this paper. It is important to note, however, that the set of consistent abstract schedules defined by the interleaving specification for a type also implicitly includes schedules in which undo operations are inserted at all possible points after an operation has been performed but prior to the end of the invoking transaction. This reflects the assumption that it must be possible to undo any operation prior to transaction commitment. As will be shown in Section 4.3, this is especially important for types that do not attempt to enforce serializability of transactions.

The specification of a shared abstract type must also include a description of how operations on behalf of multiple transactions can be interleaved. This *interleaving specification* can be used by application programmers to describe their needs to prospective type implementors or to evaluate the suitability of existing types for their applications. The specification of a shared abstract type must also list those dependencies that will be controlled to prevent cascading aborts. This part of the specification is used mainly by the type's implementor.

When specifying how operations on a shared object may interact, the amount of concurrency that can be permitted depends in part on how much detailed knowledge is available concerning the semantics of the operations [Kung 79]. We have shown how concurrency controls that distinguish those operations that only observe the state of an object ("Reads") from those that modify it ("Writes") can achieve greater concurrency than protocols not making this distinction. To increase concurrency further while still providing serializability, one can take advantage of more semantic knowledge about the operations being performed [Korth 83]. Section 4.1 illustrates how this is done in specifying Directories, using the concepts and notation of the last section.

When enough concurrency cannot be obtained even after fully exploiting the semantics of the operations on a type, it is necessary to dispense with serializability and substitute orderability with respect to some weaker set of proscribed dependency relations. Sections 4.2 and 4.3 illustrate this by comparing a serializable Queue type with a variation that preserves a weaker ordering property.

Finally, Section 4.4 discusses how implementations may reorder operations to obtain even more concurrency, and the steps that type implementors must take to demonstrate the correctness of an implementation.

4.1 Directories

As a first example, consider a Directory data type that is intended to provide a mapping between text strings and capabilities for arbitrary objects. The usual operations are provided:

- **DirInsert(dir, str, capa)**: inserts **capa** into Directory **dir** with key string **str**. Returns **ok** or **duplicate key**. The undo operation for **DirInsert** removes the inserted entry, if the insertion was successful.

- **DirDelete(dir,str)**: deletes the capability stored with key string **str** from **dir**. Returns **ok** or **not found**. The undo operation for **DirDelete** restores the deleted capability, if the deletion was successful.

- **DirLookup(dir, str)**: searches for a capability in **dir** with key string **str**. Returns the capability **capa** or **not found**. The undo operation is null, because **DirLookup** does not modify the Directory.

- **DirDump(dir)**: returns a vector of **<str,capa>** pairs with the complete contents of the Directory **dir**. The undo operation for **DirDump** is null.

Suppose one wishes to specify the Directory type so as to permit serialization of transactions that include operations on Directories. One approach would be to model each **DirInsert** or **DirDelete** operation as a Read operation followed by a Write operation, and to model each **DirLookup** or **DirDump** operation as a Read operation. The Directory type could then be specified using the Read/Write dependency relations discussed previously.

The difficulty with using such limited semantic information is that concurrency is restricted unnecessarily. For example, suppose Directories have been implemented using a standard two-phase Read/Write locking mechanism. Consider the operation **DirLookup(dir, "Foo")**, which will be blocked trying to obtain a Read lock if another transaction has performed **DirDelete(dir, "Fum")** and holds a Write lock on the Directory object. The outcome of **DirLookup(dir, "Foo")** does not depend in any way on the eventual outcome of **DirDelete(dir, "Fum")** (which may later be aborted), or vice-versa, so this blocking is unnecessary. Because **DirDelete(dir, "Fum")** may be part of an arbitrarily long transaction, the Write lock may be held for a long time and severely degrade performance.

The unnecessary loss of concurrency in this example is not the fault of this particular implementation. It is caused by the lack of semantic information in the Directory specification. By using more knowledge about the operations, this problem can be alleviated. Instead of expressing the interleaving specification for this type in terms of Read and Write operations, the type-specific Directory operations can be employed to define dependencies and the interleaving specifications can be expressed in terms of these type-specific dependencies.

To keep the number of dependencies to a minimum, the operations for the Directory data type will be divided into three groups:

- Those that modify a particular entry in the Directory. **DirInsert** and **DirDelete** operations that succeed are in this class. These are Modify (M) operations.
- Those that observe the presence, absence, or contents of a particular entry in the Directory. **DirLookup** is in this class, as are **DirInsert** and **DirDelete** operations that fail. These are Lookup (L) operations.
- Those that observe properties of the Directory that cannot be isolated to an individual entry. **DirDump** is the only operation in this class that we have defined; an operation that returned the number of entries in the Directory would also be in this class. These are Dump (D) operations.

Note that in some cases operations that fail are distinguished from those that succeed. In addition to the operations and their outcomes, the dependencies also take into account data supplied to the operations as arguments or otherwise specific to the particular object acted upon. In the following list of dependencies, the symbols σ and σ' represent distinct key string arguments to Directory operations.

The complete set of dependencies for this type is:

- D_1: $T_i{:}M(\sigma) \rightarrow T_j{:}M(\sigma')$. T_i modifies an entry with key string σ, and T_j subsequently modifies an entry with a different key string, σ'.
- D_2: $T_i{:}M(\sigma) \rightarrow T_j{:}M(\sigma)$. T_i modifies an entry with key string σ, and T_j subsequently modifies the same entry.
- D_3: $T_i{:}M(\sigma) \rightarrow T_j{:}L(\sigma')$. T_i modifies an entry with key string σ, and T_j subsequently observes an entry with a different key string, σ'.
- D_4: $T_i{:}M(\sigma) \rightarrow T_j{:}L(\sigma)$. T_i modifies an entry with key string σ, and T_j subsequently observes the same entry.
- D_5: $T_i{:}L(\sigma) \rightarrow T_j{:}L(\sigma')$. T_i observes an entry with key string σ, and T_j subsequently observes an entry with a different key string σ'.
- D_6: $T_i{:}L(\sigma) \rightarrow T_j{:}L(\sigma)$. T_i observes an entry with key string σ, and T_j subsequently observes the same entry.
- D_7: $T_i{:}L(\sigma) \rightarrow T_j{:}M(\sigma')$. T_i observes an entry with key string σ, and T_j subsequently modifies an entry with a different key string σ'.
- D_8: $T_i{:}L(\sigma) \rightarrow T_j{:}M(\sigma)$. T_i observes an entry with key string σ, and T_j subsequently modifies the same entry.
- D_9: $T_i{:}D \rightarrow T_j{:}M(\sigma)$. T_i dumps the entire contents of the Directory, and T_j subsequently modifies an entry with key string σ.
- D_{10}: $T_i{:}D \rightarrow T_j{:}L(\sigma)$. T_i dumps the entire contents of the Directory, and T_j subsequently observes an entry with key string σ.

- D_{11}: $T_i{:}M(\sigma) \rightarrow T_j{:}D$. T_i modifies an entry with key string σ, and T_j subsequently dumps the entire contents of the Directory.
- D_{12}: $T_i{:}L(\sigma) \rightarrow T_j{:}D$. T_i observes an entry with key string σ, and T_j subsequently dumps the entire contents of the Directory.
- D_{13}: $T_i{:}D \rightarrow T_j{:}D$. T_i dumps the entire contents of the Directory and T_j subsequently dumps the Directory as well.

This list is long, but it is actually quite simple to derive. There is a family of dependencies for each pair of operation classes. The key to defining the specific dependencies is the observation that when two operations refer to different strings, the relationship between the transactions that invoked them is not the same as when they refer to identical strings. Those families of dependencies for which both operation classes take a string argument therefore have two members, corresponding to these two cases. The families for which one of the operation classes is Dump have only a single member. In general, insight into the semantics of a type is needed to define the set of possible dependencies.

Like the $R \rightarrow R$ dependency, many of the Directory dependencies are insignificant and cannot affect the outcome of transactions. Hence, they may be excluded from the set of proscribed dependencies for this type. The dependencies that may be disregarded are:

- Those for which neither operation in the dependency modifies the Directory object: D_6, D_{10}, D_{12} and D_{13}. These are directly analogous to the $R \rightarrow R$ dependency.
- Those for which the two operations in the dependency refer to different key strings: D_1, D_3, D_5, and D_7.

In terms of the remaining dependencies, the interleaving specification for Directories states that an abstract schedule involving Directories is consistent if it is orderable with respect to $\{<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}\}$. The abstract Directory thus defined behaves like a collection of associatively-addressed elements, with serializability preservable independently for each element. Transactions containing operations that apply to the entire Directory, such as **DirDump**, may also be serialized, as may those that refer to multiple elements or elements that are not present.

Only two of the Directory dependencies have the potential to cause cascading aborts. These are D_4 and D_{11}. In both cases, the first operation in the dependency modifies an entry and the second operation observes that modification.

4.2 FIFO Queues

Similar specifications can be developed for other data types. The FIFO Queue provides an interesting example. We will only consider two operations:

- **QEnter(queue, capa)**: Adds an entry containing the pointer **capa** to the end of **queue**. The undo operation for **QEnter** removes this entry.

- **QRemove(queue)**: Removes the entry at the head of **queue** and returns the pointer **capa** contained therein. If **queue** is empty, the operation is blocked, and waits until **queue** becomes non-empty. The undo operation for **QRemove** restores the entry to the head of **queue**.

In order to permit serialization of transactions that contain operations on strict FIFO Queues, and to prevent cascading aborts, numerous properties must be guaranteed. For instance:

- If a transaction adds several entries to a Queue, these entries must appear together and in the same order at the head of the Queue.

- Any entries added to a Queue by a transaction may not be observed by another transaction unless the first transaction terminates successfully.

- If two transactions each make entries in two Queues, the relative ordering of the entries made by the two transactions must be the same in both Queues.

It is very easy to destroy these properties if unrestricted interleaving of operations is allowed. For instance, if QEnter operations from different transactions are interleaved, the entries made by each transaction will not appear in a block at the head of the Queue.

In defining the dependencies for the Queue type, it is necessary, as it was in the case of Directories, to distinguish individual elements in the Queue. It is assumed that each element is assigned a unique identifier[3] when it is entered on the Queue. The symbols σ and σ' are used to represent the distinct identifiers of different elements, and the **QEnter** and **QRemove** operations are abbreviated as E and R respectively. The complete set of dependencies for Queues is:

- D_1: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}E(\sigma')$. T_j enters an element σ' into the queue Q after T_i has previously entered an element σ.

- D_2: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}R(\sigma')$. T_j removes element σ' after T_i entered element σ.

- D_3: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}R(\sigma)$. T_j removes the element σ that was entered by T_i.

- D_4: $T_i{:}R(\sigma) \rightarrow_Q T_j{:}E(\sigma')$. T_j enters element σ' after T_i removed element σ.

[3] The identifier need not be globally unique, just unique among those generated for the particular Queue object.

• D_5: $T_i{:}R(\sigma) \rightarrow_Q T_j{:}R(\sigma')$. T_j removes element σ' after T_i removed element σ.

In a Read/Write synchronization scheme, QEnter must be modeled as a Write operation, and QRemove must be modeled as a Read followed by a Write. Recall that such a scheme must prevent cycles in the $<_{R\rightarrow W \,\cup\, W\rightarrow R \,\cup\, W\rightarrow W}$ dependency relation. In this case, preventing cycles in this general dependency relation is unnecessarily restrictive. Consider dependency D_2, which is formed when a transaction removes a Queue element after another transaction has previously entered a different Queue element. Neither of the transactions performing the operations can detect their ordering, nor can a third transaction. The same applies to dependency D_4, which is the inverse of D_2. As was the case for Directories, concurrency can be increased by disregarding insignificant dependencies.

To provide a strictly FIFO Queue, one must guarantee that abstract schedules are orderable with respect to the compound $<_{D_1 \cup D_3 \cup D_5}$ relation, but cycles may be permitted to form in relations that include D_2 or D_4 as long as this property is not violated. For example, consider the following schedule, in which two transactions operate on a Queue that initially contains {A, B}:

```
    T1: QEnter(Q,X)
    T2: QRemove(Q) returns A
    T1: QEnter(Q,Y)
```

At step 2 of this schedule a D_2 dependency is formed, hence $T_1 <_{D_2} T_2$. At step 3, however, a D_4 dependency is formed with $T_2 <_{D_4} T_1$. Clearly a cycle exists in the compound relation $<_{D_2 \cup D_4}$. It is easy to create other examples of consistent abstract schedules that demonstrate a cycle in the basic $<_{D_2}$ (or $<_{D_4}$) relation, or in a compound relation formed from D_2 (or D_4) together with D_1, D_3 and D_5.

The dependency relations can also be used to characterize schedules susceptible to cascading abort. Dependency relation $<_{D_1}$ is similar to the $W \rightarrow W$ dependency. Since entries made by an aborted transaction can be transparently removed from the Queue, there is no danger of cascading abort. Relations $<_{D_3}$ and $<_{D_5}$ are more similar to $W \rightarrow R$ dependencies. In a D_3 dependency, information is transferred between the transactions in the form of the queue element σ; this dependency clearly can cause cascading aborts. A D_5 dependency can also cause cascading aborts, because the removal of an element by the first transaction affects which element is received by the second transaction.

While this definition of consistency for Queues is an improvement over a Read/Write scheme, it is still very restrictive of concurrency. It allows at most two transactions, one performing QEnter operations and one performing QRemove operations, to access a Queue concurrently. Unlike the Directory, the Queue is intended to preserve a particular ordering of the elements contained in it. A system based on serializable transactions guarantees that transactions can be placed in some order; by enforcing a particular order, data types such as queues (and stacks) restrict concurrency.

4.3 Queues Allowing Greater Concurrency

The preceding examples show how the use of semantic knowledge about operations on a shared abstract type permits increased concurrency. Once such knowledge is incorporated, the limiting factor in permitting concurrency becomes knowledge about the consistency constraints that the operations in a transaction attempt to maintain [Kung 79]. This knowledge concerns the semantics of groups of operations rather than individual ones. For example, a consistency constraint might state that every Queue entry of type A is immediately followed by one of type B. The potential for such constraints was the cause of the concurrency limitations observed above.

If it is possible to restrict the consistency constraints that a programmer is free to require, types guaranteeing ordering properties weaker than serializability may be acceptable. This may permit further increases in concurrency. A variation of the queue type can be used to demonstrate this.

One of the most common uses for a queue is to provide a buffer between activities that produce and consume work. Frequently, the exact ordering of entries on the queue is not important. What is crucial is that entries put on the rear of the queue do not languish in the queue forever; they should reach the head of the queue "fairly" with respect to other entries made at about the same time. A data type having this non-starvation property can be defined: the *Weakly-FIFO Queue* (WQueue for short). A similar type, the *Semi-Queue*, has been defined by Weihl [Weihl 83b].

The operations on WQueues and their corresponding undo operations are similar to those for Queues, but the interleaving specification for WQueues allows more concurrency. The dependencies for the WQueue type are the same as for the strict Queue. However, where the strict Queue required that consistent abstract schedules be orderable with respect to $\{<_{D_1 \cup D_3 \cup D_5}\}$, the WQueue permits cycles to occur in all the dependency relations save one: $<_{D_3}$. By allowing cycles in $<_{D_1}$, the interleaving of entries by multiple transactions becomes possible. Similarly, removing D_5 from the set of proscribed dependency relations permits **WQRemove** operations to be interleaved.

To take full advantage of the greater concurrency allowed by this interleaving specification, the semantics of **WQRemove** differ slightly from those of **QRemove**. If the transaction that inserted the headmost entry in the queue has not committed, that entry cannot be removed without risking the possibility of a cascading abort. Instead, **WQRemove** scans the WQueue and removes the headmost entry for which the inserting transaction has committed. If no such element can be found, any elements inserted by the transaction doing the **WQRemove** become eligible for removal. If neither a committed entry nor one inserted by the same transaction is available, the operation is blocked until an inserting transaction commits.

Modifying the semantics of **WQRemove** in this way does not destroy the fairness properties of the WQueue. No entry will remain in the WQueue forever if:

1. The transaction that entered it commits in a finite amount of time.

2. Transactions that remove it terminate after a finite amount of time.

3. Only a finite number of transactions remove the entry and then abort.

The behavior of the WQueue is best illustrated by example. In what follows, a WQueue is represented by a sequence of letters, with the left end of the sequence being the head of the WQueue. Lower case italic letters (*a*) are used to denote entries for which the **WQEnter** operation has not committed (i.e. the transaction that performed **WQEnter** is incomplete). Upper case bold letters (**A**) are used to represent entries that have not been removed and for which the entering transaction has committed. Upper case italic letters are used for entries that have been removed by an uncommitted **WQRemove**. Superscripts on entries affected by uncommitted operations identify the transaction that performed the operation.

Assume that the WQueue is initially empty. If transactions T_1 and T_2 perform **WQEnter(WQ, a)** and **WQEnter(WQ, b)** respectively, the WQueue's state becomes:

$\{a^1, b^2\}$

Since cycles in $<_{D_1}$ are permitted, T_1 may also add another entry, yielding:

$\{a^1, b^2, c^1\}$

If T_1 and T_2 both commit, the state becomes:

$\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$

Note that the serializability of T_1 and T_2 has not been preserved. Now suppose that T_3 performs **WQRemove** and another transaction, T_4, removes two more elements:

$\{A^3, B^4, C^4\}$

If T_3 now aborts and T_4 commits, the final state becomes:

$\{\mathbf{A}\}$

In this case, A and C have effectively been reversed, even though they were inserted initially by the same transaction! This example illustrates an important difference between shared abstract types that attempt to preserve serializability and those that do not: when a type permits non-serial execution of transactions, invoking an operation and subsequently aborting it is not necessarily equivalent to not invoking the operation at all. While we do not explicitly consider the undo operations in defining dependencies or interleaving specifications, the underlying assumption that aborts can occur at any time prior to commit implies that undo operations can be inserted at any point in a schedule between the invocation of an operation and the time at which the invoking transaction commits.

Another example indicates what happens when an uncommitted entry reaches the head of the Queue. Suppose the initial state is:

$\{a^5, b^6\}$

If T_6 commits but T_5 remains incomplete, the state becomes:

$\{a^5, B\}$

If T_7 removes an element at this time, B will be returned, leaving:

$\{a^5\}$

after T_7 commits. On the other hand, if T_5 commits after T_6, but before the remove by T_7, A will be returned even though its insertion was committed after B's.

To summarize the comparison between the WQueue and the ordinary Queue, note that two properties of the regular Queue have been sacrificed. First, strict FIFO ordering of entries is not guaranteed, because aborting **WQRemove** operations can reorder them. Second, transactions that operate on WQueues are not necessarily serializable with respect to all transactions in the system. Some other crucial properties, however, are preserved. The WQueue will not starve any entry, and it enforces an ordering of those transactions that communicate through access to a common element of the queue. This is ensured by orderability with respect to $\{<_{D_3}\}$. These modifications greatly increase concurrency, while still providing a data type that is useful in many situations.

4.4 Proving the Correctness of Type Implementations

Whereas the user of a type may employ the specified properties of abstract schedules (along with the rest of the type's specification) to reason about the correctness of transactions, the implementor of a type must prove the correctness of an implementation given the order in which operations are actually invoked. Real implementations may reorder the operations on an object to improve concurrency without changing the type's interleaving specification. Consider an implementation of the Queue type in which elements to be entered by a transaction are first collected in a transaction-local cache and entered as a block at end-of-transaction. This implementation allows any number of transactions to invoke the **QEnter** operation simultaneously, provided care is taken to serialize correctly transactions involving multiple Queues. By actually performing the insertions as a block, this implementation effectively reorders the individual **QEnter** operations to preserve consistency. It is possible to reorder **QEnter** operations in this way because **QEnter** does not return any information to its caller. Formation of any dependencies that might result from its invocation can therefore be postponed. The ultimate ordering of operations in the abstract schedule is determined by the implementation once all the **QEnter** operations to be performed by a given transaction are known. Thus, this implementation has the benefit of more knowledge about transactions than has the standard implementation.

Invocation schedules list operations in the order in which they are actually invoked, rather than in order of their abstract effects[4]. For example, the following is a possible invocation schedule for a Queue implemented using the block-insertion technique described above:

```
T2: QEnter(Q,Y)
T1: QEnter(Q,X)
T3: QRemove(Q)
```

If T_1 commits before T_2, the implementation reorders the two QEnter operations, resulting in the abstract schedule:

```
T1: QEnter(Q, X)
T2: QEnter(Q, Y)
T3: QRemove(Q)
```

The mapping between invocation schedules and abstract schedules is many-one; each invocation schedule implements exactly one abstract schedule, but an abstract schedule may be implemented by multiple invocation schedules. The synchronization mechanism used by an implementation determines a set of invocation schedules, called *legal schedules*, that are permitted by the implementation. The implementor must show that all legal invocation schedules map to consistent abstract schedules. To prevent cascading aborts as well, implementors must use a synchronization strategy that restricts the set of legal invocation schedules to those that map to abstract schedules that are in the intersection of the consistent and cascade-free sets.

5 Orderability of Groups of Transactions

The preceding section described how the standard specification of an abstract type, which only seeks to characterize the type's invariants and the postconditions for its operations, can be augmented with an interleaving specification that describes the local synchronization properties of objects. In this section we broaden our focus from the properties of the typed objects that are manipulated by transactions to the properties of entire transactions. We first examine how to generalize the definition of consistent abstract schedules to schedules that include operations on more than one object type, and then consider how ordering properties of groups of transactions can be used to show their correctness.

[4]It is assumed that the actual concurrent execution of the transactions can be modeled by a linear ordering of their component operations. This requires that the primitive operations be (abstractly) atomic. In the multiprocessor case, all linearizations of operations that could occur simultaneously yield distinct invocation schedules.

5.1 How the Specifications of Multiple Types Interact

Guaranteeing orderability with respect to the proscribed relations of a collection of individual types is not sufficient to ensure global ordering properties of transactions, such as serializability. Consider the following schedule, which contains transactions that operate both on Queues and Directories. Each of these types preserves orderability with respect to the union of all significant dependencies for the individual type, in order that transactions involving the type may potentially be serialized. However, this property alone does not guarantee serializability of the transactions. For example, the following schedule is not serializable:

```
T1: QEnter(Q,X)
T2: QEnter(Q,Y)
T2: DirInsert(D, "A", Z)
T1: DirDelete(D, "A")
```

Let $<_{Dir}$ stand for the $<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}$ relation, defined earlier for type Directory. Let $<_Q$ stand for the $<_{D_1 \cup D_3 \cup D_5}$ relation, defined earlier for Queues. Although the schedule is orderable with respect to $\{<_{Dir}, <_Q\}$, it is not serializable. To achieve serializability, the Queue and Directory types must cooperate to prevent cycles in the relation $\{<_{Dir \cup Q}\}$. The schedule is not orderable with respect to this compound dependency.

This example indicates how to generalize the definition of consistency to apply to abstract schedules containing operations on multiple types. Assume the interleaving specification for type Y_1 guarantees orderability with respect to $\{<_{D_1}\}$, the interleaving specification for type Y_2 guarantees orderability with respect to $\{<_{D_2}\}$, etc. The set of consistent abstract schedules involving types $Y_1, Y_2, \ldots Y_n$ is defined as those abstract schedules that are orderable with respect to $\{<_{D_1 \cup D_2 \cup \ldots \cup D_n}\}$: the union of the proscribed dependency relations of the individual types. A set of types whose implementations satisfy this property is called a set of *cooperative* types.

The need for cooperation among types does not necessarily imply that whenever a system is extended by the definition of a new type, the synchronization requirements of all existing types must be rethought. When designing a system, however, the implementors of cooperative types must first agree on a synchronization mechanism that is sufficiently flexible and powerful to meet all of their requirements. A poor choice of mechanism for fundamental building-block types will have an adverse effect on the entire system. Section 6 describes a mechanism based on locking that permits highly concurrent implementations of a large variety of shared abstract types.

5.2 Correctness of Transactions

When all of the types involved in a group of transactions cooperate to preserve an ordering property equivalent to serializability, it is easy to show that the correctness of transactions is not affected by concurrency. Because transactions are completely isolated from one another, a transaction can be proven correct solely on the basis of its own code and the assumption that the system state is correct when the transaction is initiated.

It is much more difficult to prove the correctness of transactions when they include operations on types that permit non-serializable interaction among transactions. One must consider the possible effects of interleaving each transaction with any other transaction, subject to the constraints of whatever ordering property is guaranteed by the collection of types. Nevertheless, in many practical situations, this task should not be insurmountable. We give two examples of situations where it is possible to make useful inferences about the behavior of transactions even though they preserve an ordering property weaker than serializability.

Users often invoke the **DirDump** operation on a Directory when they are "just looking around." In such cases, users would like to see a snapshot of the Directory's contents at an instant when the status of each entry is well defined, but they don't care what happens to the Directory thereafter. If all Directory operations attempt to enforce serializability, using **DirDump** in this way could greatly restrict concurrency. This problem can be alleviated by modifying the specification of the Directory type to permit limited non-serializable behavior.

Suppose dependency relations containing D_9: T_i:D $\rightarrow$ T_j:M(σ) are removed from the set of proscribed relations for the modified Directory type. That is, the interleaving specification for Directories only requires orderability with respect to $\{<_{D_2 \cup D_4 \cup D_8 \cup D_{11}}\}$ instead of $\{<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}\}$. Although this modified Directory allows non-serializable behavior, one can still guarantee that certain consistency constraints are not violated. For example, if a transaction replaces a group of entries in a Directory, one can still prove that no other transaction doing **DirLookup** operations will observe an incompatible collection of entries.

The WQueue of section 4.3 provides another example of a useful type that permits non-serializable interaction of transactions. Although the ordering property for WQueues is weaker than the one for strict Queues, some interesting properties can still be deduced based only on orderability with respect to $\{<_{D_3}\}$. Consider two transactions, T_1 and T_2, and two WQueues, Q_1 and Q_2. Suppose T_1 is intended to move all elements from Q_1 to Q_2 and T_2 is intended to move all elements from Q_2 to Q_1. If these transactions are run concurrently, the elements should all wind up in one WQueue or the other. This can be guaranteed only if $<_{D_3}$ is proscribed; otherwise elements could be shuffled endlessly between Q_1 and Q_2 and the transactions might never terminate.

6 A Technique for Synchronizing Shared Abstract Types

We have developed a formalism for specifying the synchronization of operations on shared abstract types, and interleaving specifications for some example types have been given. This section outlines a synchronization mechanism that can be used in implementations of these types. While we do not describe a particular syntax or implementation for this mechanism, we show how it can be used to prevent cascading aborts and control the interleaving of operations. We show how it provides the cooperation among types that is needed to preserve serializability or a weaker ordering property of a group of transactions. Implementation sketches for the shared abstract types specified in Section 4 are given as examples of its use.

As indicated in Section 4.4. the implementor of a type must take the following steps to demonstrate the correctness of an implementation:

1. characterize the set of legal invocation schedules, that is, those invocation schedules allowed by the synchronization mechanism used in the implementation.
2. give a mapping from invocation schedules to abstract schedules, and prove that the implementation carries out this mapping.
3. prove that every legal invocation schedule yields a consistent abstract schedule under this mapping.

This three-part task is simplest for implementations that are idealized in that they do not reorder operations on objects. Under these conditions, invocation schedules and abstract schedules are equivalent, and the second step in this process can be eliminated. The examples in this section discuss such idealized implementations of types.

6.1 Type-Specific Locking

The proposed synchronization technique is based on *locking*, which is used in many database systems to synchronize access to database objects. There are many variations on locking, but the same basic principle underlies them all: before a transaction is permitted to manipulate an object, it must obtain a *lock* on the object that will restrict further access to the object by other transactions until the transaction holding the lock releases it.

Locking restricts the formation of dependencies between transactions by restricting the set of legal invocation schedules. Whenever one transaction is forced to wait for a lock held by another, the formation of a dependency between the two transactions is delayed until the first transaction releases the lock. Under the well-known *two-phase locking* protocol [Eswaran 76], no transaction releases a lock until it has already claimed all the locks it will ever claim. This has the effect of converting potential cycles in dependency relations into deadlocks instead. These can be detected, and because no dependencies have yet been allowed to form, either transaction can be aborted without affecting the other.

Locking is a conservative policy, because it delays the formation of any dependency that is part of a proscribed relation, not just those that eventually lead to cycles. This is not as significant a disadvantage as it might appear, however, because formation of those dependencies that transfer information (see Section 3.3) must be delayed anyway to prevent cascading aborts. In fact, the even more restrictive strategy of holding certain locks until end-of-transaction must often be employed to ensure that schedules are cascade-free. Furthermore, it is the conservative nature of locking protocols that makes them a suitable mechanism for sets of cooperative types. By preventing the formation of any dependencies local to a single object, cycles in proscribed relations that involve multiple types are automatically avoided without explicit communication between type managers. This is an important advantage, because it allows type managers to be constructed independently, as long as they correctly prevent the local formation of dependencies.

The chief disadvantage of many locking mechanisms is that they sacrifice concurrency by making minimal use of semantic knowledge about the objects being manipulated. The simplest locking schemes use only one type of lock, and hence cannot distinguish between significant and insignificant dependencies. Read/Write locking schemes use some semantic information, but are not flexible enough to take advantage of the extra concurrency specifiable in terms of type-specific dependencies. It has been shown [Kung 79] that two-phase locking is optimal under such conditions of limited semantic knowledge, but much more concurrency can be obtained if more semantic information is used. The locking technique described here generalizes the ideas behind Read/Write locking. It permits the definition of type-specific locking rules that reflect the interleaving specifications of individual data types. More restrictive type-specific locking schemes have previously been investigated by Korth [Korth 83].

Two observations can be made concerning type-specific dependencies. First, they specify the way in which type-specific operations on behalf of different transactions may be interleaved. Analogously, the generalized locking scheme requires the definition of type-specific *lock classes*, which correspond roughly to the operations on the type. Second, in addition to the operations, the dependencies reflect data supplied to the operations as arguments or data that is otherwise specific to the particular object acted upon. Therefore, an instance of a lock in the generalized locking scheme consists of two parts: the type-specific lock class and some amount of instance-specific data. It is the inclusion of data in the lock instance that differentiates our technique from Korth's. We use the notation **{LockClass(data)}** to represent an instance of a lock.

Once the lock classes for a type have been defined, a Boolean function must be given that specifies whether a particular new lock request may be granted as a function of those locks already held on the object. In accordance with the practice in database literature, this function will be represented by a *lock compatibility table*. Only those locks held by other transactions need be checked for compatibility; a new lock request is always compatible with other locks held by the same transaction.

To complete the description of a type's locking scheme, one must specify the protocol by which each of the type's operations acquires and releases locks. Although two-phase locking can be used with type-specific locks, the locking protocol may also be type-specific. A uniform two-phase protocol is simplest to understand, but the added flexibility of type-specific protocols can allow increased concurrency. The exact nature of a type-specific protocol depends not only on the semantics of the type, but also on the particular representation and implementation chosen.

6.2 Directories

A simple idealized implementation of the Directory type specified in Section 4.1 illustrates the basics of type-specific locking. In this example, it is assumed that the Directory operations have been implemented in a straightforward fashion with no attempt at internal concurrency. It is further assumed that the operations act under the protection of a monitor or other mutual exclusion mechanism during the actual manipulation of Directory objects. Locking is used exclusively to control the sequencing of Directory operations on behalf of multiple transactions. The locking and mutual exclusion mechanisms cannot be completely independent, however, because mutual exclusion must be released when waiting for a lock within the monitor. This is a standard technique in systems that use monitors for synchronization [Hoare 74].

Because the mapping from invocation schedules to abstract schedules is trivial for this implementation, the second step of the validation process is eliminated. The discussion of the locking scheme for Directories therefore focuses on the first and third steps: informal characterization of the set of legal schedules, and comparison of this set with the set of consistent schedules.

As was noted in Section 4.1, the operations for the Directory data type can be divided into three groups:

- Modify operations, that alter the particular Directory entry identified by the key string σ.
- Lookup operations, that observe the presence, absence, or contents of the particular Directory entry identified by the key string σ.
- Dump operations, that observe properties of the Directory that cannot be isolated to an individual entry.

Corresponding to these groups, three lock classes can be defined:

- **{DirModify(σ)}**: To indicate that an incomplete transaction has inserted or deleted an entry with key string σ.

- **{DirLookup(σ)}**: To indicate that an incomplete transaction has attempted to observe the entry with key string σ.

- **{DirDump}**: To indicate that an incomplete transaction has performed a **DirDump** of the entire directory.

The lock compatibility table for Directories can be found in Table 1. Since there are a potentially infinite number of strings, the symbols σ and σ' are used to represent two arbitrary non-identical strings.

		Lock Held		
		DirModify(σ)	**DirLookup(σ)**	**DirDump**
Lock Requested	**DirModify(σ)**	No	No	No
	DirModify(σ')	OK	OK	No
	DirLookup(σ)	No	OK	OK
	DirLookup(σ')	OK	OK	OK
	DirDump	No	OK	OK

Table 1: Lock Compatibility Table for Directories

Each entry in this table reflects the nature of one of the type-specific dependency relations for Directories. Compatible entries represent dependency relations in which cycles are allowed to occur: for example, the entry in row 2, column 2 is "OK" because cycles are permitted in the $<_{M(\sigma) \rightarrow M(\sigma')}$ dependency relation. Incompatible entries reflect proscribed relations, such as the entry in row 1, column 2, which is due to the proscribed $<_{M(\sigma) \rightarrow M(\sigma)}$ relation.

The protocol used by the Directory operations for acquiring and releasing locks is as follows:

- **DirInsert** or **DirDelete** operations that specify the key string σ obtain a **{DirModify(σ)}** lock on the Directory. If the operation succeeds, the lock is held until end-of-transaction. If the operation fails, the lock is converted to a **{DirLookup(σ)}** lock, which is held until end-of-transaction.

- **DirLookup** operations that specify the key string σ obtain a **{DirLookup(σ)}** lock on the Directory that is held until end-of-transaction.

- **DirDump** operations obtain a **{DirDump}** lock on the Directory that is held until end-of-transaction.

The following example demonstrates how the components of the locking scheme interact. Suppose a Directory D is initially empty. If a transaction T_1 performs the operation **DirDelete(D, "Zebra")**, this operation will fail by returning **not found** and leave a **{DirLookup("Zebra")}** lock on the Directory until the termination of T_1. Now suppose a second transaction, T_2, performs the operation **DirInsert(D, "Zebra", capa)**. According to the protocol, **DirInsert** must first obtain a **{DirModify("Zebra")}** lock. Because the dependency relation $<_{L(\sigma) \rightarrow M(\sigma)}$ is proscribed, this lock is incompatible with the

{DirLookup("Zebra")} lock already held by T_1 (see row 1, column 3 of the compatibility table). Therefore, T_2 will be blocked. If T_1 subsequently becomes blocked while attempting to access an object already locked by T_2, a deadlock will occur. Both transactions are then blocked attempting to form dependencies that are part of proscribed relations. Although these relations may involve different objects, or even different types, a cycle in the union of the two relations is effectively prevented. This is exactly the behavior required to achieve consistency among cooperative types. On the other hand, if T_1 completes successfully the lock is released and the dependency of T_2 on T_1 is permitted to form. Since the $L(\sigma) \rightarrow M(\sigma)$ dependency cannot lead to cascading aborts, one may conclude (after the fact) that delaying T_2 was unnecessary.

By contrast, a transaction T_3 that performs the operation **DirInsert(D, "Giraffe",capa)** need not be blocked, because the $<_{L(\sigma)\ \rightarrow\ M(\sigma')}$ dependency relation is not proscribed. Accordingly, row 2, column 3 of the compatibility table indicates that a **{DirModify("Giraffe")}** lock is compatible with a **{DirLookup("Zebra")}** lock.

Although not a formal proof, this example characterizes the set of legal schedules permitted by the implementation, and shows how the lock classes, compatibility table, and locking protocol combine to guarantee that the legal schedules correspond to the consistent schedules defined in the last section. They capture the idea that, for this abstract data type, synchronization of access depends on the operations being performed, the particular entries in the Directory they attempt to reference, and their outcome. Because locks are on Directory objects, not components of directories, the technique also handles phantoms: entries that are mentioned in operations but are not present in the Directory.

6.3 Strictly FIFO Queues

Type-specific locking can also be used in implementations of the Queue data type of Section 4.2. As in the preceding example, assume a idealized implementation operating under conditions of mutual exclusion. To implement strictly FIFO Queues supporting only **QEnter** and **QRemove** operations, two lock classes are sufficient: **{QEnter(σ)}** and **{QRemove(σ)}**. As in the case of Directories, locks on Queues identify the particular entry to which the operation requesting the lock refers. Since Queue entries are not identified by key strings, it is assumed that at QEnter time, each element is assigned an identifier unique to the Queue instance. These identifiers correspond to those used in defining the dependency relations. Thus, a **{QEnter(σ)}** lock indicates that an element with identifier σ has been entered into the Queue by an incomplete transaction. Likewise, a **{QRemove(σ)}** lock indicates that the element with identifier σ has been removed form the Queue by an incomplete transaction.

The protocol for the Queue operations is:

- QEnter operations must obtain a {QEnter(σ)} lock, where σ is the newly-assigned identifier for the entry to be added. This lock is held until end-of-transaction.

- QRemove operations must obtain a **{QRemove(σ)}** lock, where σ is the identifier of the entry at the head of the Queue. This lock is held until end-of-transaction. Note that obtaining a **{QRemove(σ)}** lock does not necessarily imply that an entry σ is actually in the Queue, because the transaction that made the entry may have since aborted. If so, the **QRemove** operation must request a **{QRemove(σ')}** lock on the new headmost entry, σ'.

Table 2 shows the lock compatibility table for Queues. As usual, the symbols σ and σ' represent the identifiers of two different elements. Because the element identifiers are unique, certain situations (e.g. attempting to enter an element with the same identifier as an element already removed) cannot occur. The compatibility function is undefined in these cases, so the table entries are marked 'NA' for 'Not Applicable'.

		Lock Held	
		QEnter(σ)	**QRemove(σ)**
Lock Requested	**QEnter(σ)**	NA	NA
	QEnter(σ')	No	OK
	QRemove(σ)	No	NA
	QRemove(σ')	OK	No

Table 2: Lock Compatibility Table for Queues

The lock compatibility table reflects the limited concurrency of this type. Once a QRemove operation has retrieved the entry with identifier σ, some entry with identifier σ' becomes the head element of the Queue. But other transactions will be blocked trying to obtain the **{QRemove(σ')}** lock needed to remove it, until the first transaction completes. Multiple **QEnter** operations on behalf of different transactions interact in the same way. The incompatibility of **{QRemove(σ)}** with **{QEnter(σ)}** ensures that an uncommitted entry cannot be removed from the Queue, thereby eliminating a potential cause of cascading aborts.

6.4 WQueues

For a comparable idealized implementation of WQueues supporting only **WQEnter** and **WQRemove**, the same lock classes may be used as for FIFO Queues. The major difference between the two types shows up in the lock compatibility function, given by Table 3. To reflect the allowability of interleaved **WQEnter** operations by different transactions, the table entry in row 2, column 2 defines **{WQEnter(σ)}** and **{WQEnter(σ')}** locks to be compatible. Similarly, the entry in row 4, column 3 now permits multiple transactions to perform **WQRemove** operations. The only remaining restriction is the one in row 3, column 2 that prevents uncommitted entries from being removed. This prevents cycles in the proscribed $<_{E(\sigma) \rightarrow R(\sigma)}$ dependency relation and, because the lock is held until end-of-transaction, also prevents cascading aborts.

		Lock Held	
		WQEnter(σ)	**WQRemove(σ)**
Lock Requested	**WQEnter(σ)**	NA	NA
	WQEnter(σ')	OK	OK
	WQRemove(σ)	No	NA
	WQRemove(σ')	OK	OK

Table 3: Lock Compatibility Table for WQueues

The locking protocol for the WQueue operations is substantially the same as the one for the Queue operations. The only difference is that a **WQRemove** operation that is unable to obtain the required **{WQRemove(σ)}** lock on the element at the head of the WQueue does not block. Instead, **WQRemove** searches down the WQueue for some other element with identifier σ', for which a **{WQRemove(σ')}** lock can be obtained. This reflects the property of WQueues that permits elements farther down the WQueue to be removed when the head element is uncommitted. If no element can be found, the operation is blocked until an inserting transaction commits.

6.5 Summary

The examples in this section have shown how type-specific locking can be used for synchronization in implementations of several data types. The examples show how locking can be used to prevent cycles in proscribed dependency relations, including cycles containing several types of objects. They also indicate how locking can be used to prevent cascading aborts.

A full discussion of the syntax and implementation of type-specific locking mechanisms is beyond the scope of this paper. Further work is needed to determine the specific primitives required for definition of new object types, locking, unlocking, conditional locking, etc. Another area requiring further study is the relationship between the locking mechanism and other synchronization mechanisms that are used for mutual exclusion and to signal events. It appears, however, that implementation of a type-specific locking mechanism is often no more complex or expensive than implementations of standard locking. Unlike predicate locking schemes [Eswaran 76], the set of locks that apply to a particular object can easily be determined. It is also not difficult to determine what processes may be awakened in response to an event such as transaction completion.

7 Summary

This paper has been concerned with synchronizing transactions that access shared abstract types. In our model, four properties distinguish such types from others:

- Operations on them are permanent.
- They support failure atomicity of transactions.
- They do not permit cascading aborts.
- They contribute to preserving ordering properties of groups of transactions.

These properties are not independent, and the mechanisms that are used to achieve them are therefore related as well.

Schedules and dependencies are useful in understanding the interaction between concurrent transactions. The well-known consistency property of serializability can be redefined as a special case of orderability with respect to a dependency relation. The specific dependency relation depends on how much semantic knowledge is available concerning operations on objects. When Read operations are distinguished from Write operations, serializability requires orderability with respect to a less restrictive dependency relation than when this distinction is not made. Dependencies can also be used to characterize schedules that are not prone to cascading aborts.

Additional type-specific semantic knowledge about operations can allow additional concurrency. The interleaving specifications for Directories and Queues developed in Sections 4.1 and 4.2 were stated in terms of orderability with respect to type-specific dependencies. To increase concurrency further, the WQueue sacrifices serializability while preserving orderability with respect to a less restrictive dependency. When several abstract types are combined in a transaction, orderability must be guaranteed with respect to the relation that is the union of the proscribed relations of the individual types.

Section 6 described a locking mechanism for implementing the synchronization required by the types described in Section 4. By allowing locks that consist of a type-specific lock class and instance-specific data, the mechanism provides a powerful framework for using type-specific semantics in synchronization. This mechanism is suitable for use in transactions containing multiple types, and it can also be used to prevent cascading aborts. The implementation of Directories shows how type-specific locking permits a uniform treatment of the problem of phantoms. Locks need not be directly associated with particular components of objects, which facilitates the separation of synchronization from other type representation issues. The examples of various Queue types show the mechanism's flexibility.

This paper has not provided a complete discussion of the issues involved in the specification and implementation of shared abstract types. For example, we have not discussed the construction of compound shared abstract types, which use other shared abstract types in their implementation. (However, Schwarz [Schwarz 82] gives an example of this.) In addition, we have hardly mentioned recovery considerations, though we believe logging mechanisms as described by Lindsay [Lindsay 79] can be extended to meet the needs of shared abstract types. Recovery is discussed more fully in a related paper [Schwarz 83]. Finally, we have not discussed specific algorithms for coping with deadlocks.

Clearly, the definition and implementation of shared abstract types is more difficult than the definition and implementation of regular abstract types. However, once these types are implemented, programmers can construct arbitrary transactions that invoke operations on the types. These transactions should greatly simplify the construction of reliable distributed systems. Though this paper has focused entirely on synchronization, we believe that this topic is central to understanding how transactions can be used as a basic building block in the implementation of distributed systems.

8 Acknowledgments

We gratefully acknowledge the helpful technical and editorial comments that were made by Dean Daniels, Cynthia Hibbard, Andy Hisgen, Bruce Lindsay, and Irving Traiger. We also gratefully acknowledge the Archons Project at Carnegie-Mellon University for providing the framework in which we conducted this research.

References

[Allchin 83] J. E. Allchin, M.S. McKendry.
Synchronization and Recovery of Actions.
In *Proc. of the Second Principles of Distributed Computing Conference*, pages 31-44. August, 1983.

[Bernstein 79] Philip A. Bernstein, David W. Shipman and Wing S. Wong.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering SE-5(3):203-216, May, 1979.

[Bernstein 81] Philip A. Bernestein and Nathan Goodman.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[Dahl 72] O.-J. Dahl and C. A. R. Hoare.
Hierarchical Program Structures.
In C. A. R. Hoare (editor), *A.P.I.C. Studies in Data Processing.* Volume 8: *Structured Programming*, chapter 3, pages 175-220. Academic Press, London and New York, 1972.

[Dept. of Defense 82]
Reference Manual for the Ada Programming Language
July 1982 edition, Dept. of Defense, Ada Joint Program Office, Washington, DC, 1982.

[Eswaran 76] K. P. Eswaran, James N. Gray, Raymond A. Lorie, I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Comm. of the ACM 19(11), November, 1976.

[Garcia-Molina 83]
Hector Garcia-Molina.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transactions on Database Systems 8(2):186-213, June, 1983.

[Gray 75] J. N. Gray, R.A. Lorie, G. R. Putzolu, and I. L. Traiger.
Granularity of Locks and Degrees of Consistency in a Shared Data Base.
IBM Research Report RJ1654, IBM Research Laboratory, San Jose, Ca., September, 1975.

[Gray 80] Jim Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, Ca., August, 1980.

[Gray 81] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys (2):223-242, June, 1981.

[Hoare 74] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Comm. of the ACM 17(10):549-557, October, 1974.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), Jaunary, 1983.

[Kung 79] H. T. Kung and C. H. Papadimitriou.
An Optimality Theory of Concurrency Control for Databases.
In *Proceedings of the 1979 SIGMOD Conference.* ACM, Boston, MA., May, 1979.

[Lampson 81] Butler W. Lampson.
Atomic Transactions.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science.* Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 11, pages 246-265. Springer-Verlag, 1981.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

[Liskov 77] B. Liskov, A. Snyder, R.Atkinson, C. Schaffert.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8), August, 1977.

[Liskov 82a] Barbara Liskov.
On Linguistic Support for Distributed Programs.
IEEE Trans. on Software Engineering SE-8(3):203-210, May, 1982.

[Liskov 82b] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 7-19. Albuquerque, NM, January, 1982.

[Lomet 77] David B. Lomet.
Process Structuring, Synchronization, and Recovery Using Atomic Actions.
ACM SIGPLAN Notices 12(3), March, 1977.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.

[Papadimitriou 77]
C. H. Papadimitriou, P. A. Bernstein, and J. B. Rothnie.
Some Computational Problems Related to Database Concurrency Control.
In *Proceedings of the Conference on Theoretical Computer Science.* Waterloo, Ont., August, 1977.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.

[Schwarz 82] Peter M. Schwarz.
Building Systems Based on Atomic Transactions.
1982.
PhD. Thesis Proposal, Carnegie-Mellon University.

[Schwarz 83]	Peter M. Schwarz, Alfred Z. Spector. *Recovery of Shared Abstract Types.* Carnegie-Mellon Report CMU-CS-83-151, Carnegie-Mellon University, Pittsburgh, PA, October, 1983.
[Sha 83]	Lui Sha, E. Douglas Jensen, Richard F. Rashid, J. Duane Northcutt. Distributed Co-operating Processes and Transactions. In *Proceedings ACM SIGCOMM Symposium.* 1983.
[Weihl 83a]	William E. Weihl. data Dependent Concurrency Control and Recovery. In *Proc. of the Second Principles of Distributed Computing Conference*, pages 73-74. August, 1983.
[Weihl 83b]	W. Weihl, B. Liskov. Specification and Implementation of Resilient, Atomic Data Types. In *Symposium on Programming Language Issues in Software Systems.* June, 1983.
[Wulf 74]	W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack. HYDRA: The Kernel of a Muiltiprocessor Operating System. *Communications of the ACM* 17(6):337-345, June, 1974.
[Wulf 76]	W. A. Wulf, R. L. London, M. Shaw. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering* SE-2(4), December, 1976.