Concurrency Control In
Resilient Objects*

Kenneth P. Birman
Thomas A. Joseph
Thomas Räuchle

TR 84-622
July 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This research has been supported in part by a grant from the Sperry Corporation.
A short version of this report has been published under the title "Extending Resilient Objects Efficiently" in the proceedings of the 2nd GI/NTG/GMR Conference on Fault-Tolerant Computing Systems, Bonn, 1984.
CONCURRENCY CONTROL IN RESILIENT OBJECTS

Kenneth P. Birman, Thomas A. Joseph, Thomas Räuchle

Department of Computer Science
Cornell University, Ithaca, New York, 14853

ABSTRACT
Resilient objects are instances of distributed abstract data types that are tolerant to failures. Due to the distributed nature of resilient objects and the use of replicated data, the potential for a high degree of concurrency exists within them. This paper introduces a new concurrency control algorithm which achieves higher concurrency than conventional methods like two-phase locking. Objects are specified in a high level language. The algorithm uses the specification taking advantage of the structure of resilient objects and exploiting semantic information about operations.

1. Introduction

ISIS is a new project that seeks to develop a programming environment for designing highly available distributed systems. ISIS is based on the idea of constructing fault-tolerant distributed systems from k-resilient objects. Such objects are instances of distributed abstract data types that are capable of remaining correct and available despite k site failures; they also guarantee that an operation proceeds to completion despite up to k site failures. A k-resilient object must reside at more than k sites to guarantee that after k site failures there is still at least one more site at which a component of the object provides access to data and is able to continue execution. Consistency of the copies of the object data is ensured by using techniques developed for distributed data bases [Ber83][Ske81].

Operations on resilient objects can be executed concurrently. This necessitates a concurrency control mechanism, which guarantees that operations are executed atomically, i.e., they are serialized with respect to each other. The structure of interactions between components of resilient objects makes a high degree of concurrency very desirable. In contrast to traditional data base environments, information about the structure of data, objects, and about the semantics of operations is available in the ISIS environment. The ISIS scheduling mechanism exploits this information to achieve a high degree of concurrency without introducing a large message overhead. The scheduling mechanism consists of two parts, the scheduler, which incorporates the concurrency control

1 This research has been supported in part by a grant from the Sperry Corporation.

A short version of this report has been published under the title "Extending Resilient Objects Efficiently" in the proceedings of the 2nd GI/NTG/GMR Conference on Fault-Tolerant Computing Systems, Bonn, 1984

2 Tolerated failures are stopping failures, that is, sites fail by stopping any activity. An underlying failure detector periodically probes the sites to detect failures and presents the abstraction of fail-stop processors[Sch83].
mechanism, and the executor, which applies data flow analysis techniques to optimize communication between the components of an object. In this paper we describe the scheduler; the executor will be described in a forthcoming paper. The proposed concurrency control method achieves higher concurrency than two-phase locking, while avoiding some of the shortcomings of optimistic concurrency control schemes.

The paper is organized as follows: Section 2 discusses resilient objects and defines the concurrency control problem arising within resilient objects. In section 3 a centralized version of the ISIS concurrency control algorithm is described; a distributed version is presented in section 4. Section 5 discusses fault tolerance issues. Conclusions are presented in section 6.

2. Resilient Objects

2.1. Operations on Objects

Resilient objects are instances of resilient abstract data types. They exist at multiple sites and are accessed through a remote procedure call mechanism that has been extended to redirect messages to operational sites in case of failure of the components of the sender or the receiver. Resilient objects encapsulate data, which are accessed by requests at the user interface. Object types can also be constructed by combining other object types, or by extending the interface of existing types, or a combination of these techniques. In these cases, an object performing an operation could invoke an operation in another object, giving rise to nested operations. The requirement that operations be atomic makes them similar to nested transactions [Mos81].

When an object is invoked to perform an operation, one of its components is designated the coordinator for this request. The other components are its cohorts. The coordinator supervises the execution of the request, which consists of one or more suboperations. While executing the operation, the coordinator transmits information to its cohorts to ensure that the operation can be completed should the coordinator fail at some point during execution [Bir84a]. To serialize execution of operations, the coordinator first passes an internal representation of an operation to the serializer, together with the arguments of the operation (Figure 1). The serializers at the different sites execute the protocol described in section 4 to determine a serialization order for the request. The executor then groups the operations into "meta-operations" to minimize interactions between components and distributes the meta-operations to the coordinators. The coordinators then execute the operation in an order consistent with the serialization order.

The serializer uses a new concurrency control algorithm that differs from other algorithms developed for nested transactions [Mos81][Lis83][Lyn82]. The standard algorithm, two-phase locking, is easy to implement, but represents a conservative approach to concurrency control: except in case of deadlocks, transactions do not have to be aborted in order to guarantee serializability. It is well known, however, that two-phase locking does not achieve maximum concurrency. Optimistic methods have been proposed[Kun81], but they are only suited for environments where the interaction
between transactions is infrequent. Otherwise these algorithms can give rise to "cascading aborts": the phenomenon where aborting one transaction necessitates aborting other transactions that could have read its results, thus introducing serious overhead.

Our algorithm reflects a compromise between these two approaches. It attains greater concurrency than two-phase locking, while at the same time controlling the overhead of cascading aborts.

2.2. Dependencies Between Operations

A concurrency control algorithm must respect certain dependencies between operations. The definition of an operation introduces data dependencies and precedence dependencies between its suboperations.

A data dependency exists between two operations when one of them requires the result of the other before it can be executed. In the program below data dependencies exist between operations 1 and 3, and 2 and 3, but not between 1 and 2.

Example 1:

1: read(x);
2: read(y);
3: z = x + y;

A nested operation has a data dependency on each suboperation whose results it requires to compute its own result. If an operation has side-effects, it may not have a data dependency on those suboperations that create the side-effect; consequently, its result may be available before those suboperations are completed.

Precedence dependencies exist between conflicting operations. Intuitively, operations conflict, and thus must be serialized, if they are accessing the same item and the value of the item reflects the order of access, i.e., from the value of the item it can be
deduced in which order the transactions accessed it. The dependencies arise from two sources: they are either part of the specification, or are introduced by the concurrency control mechanism as described in the next section. A precedence dependency exists between operations $A$ and $B$ if $B$ has no data dependency on $A$, but depends on some side effect of $A$. $A$ and $B$ can then be executed concurrently, provided $B$ is serialized after $A$. For example, in an event scheduling mechanism, the execution of an event may generate new events to be scheduled in the future. Typical operations then are inserting a newly created event into a priority queue and deleting the next event to occur from the queue. These operations can be executed in parallel since there is no data dependency between them. However, the insertion of a new event must be serialized before the deletion of the next event, in case the queue is initially empty and, consequently, the element to be inserted is identical with the next event to be deleted.

For nested operations, serializing operation $A$ before operation $B$ implies that all suboperations of $A$ must also be serialized before those of $B$. Furthermore, because operations are atomic, $parent(A)$ must be serialized before $parent(B)$. Thus, a precedence dependency between $A$ and $B$ results in a precedence dependency between the highest distinct ancestors of $A$ and $B$.

Both data and precedence dependencies are inherent in ISIS object specifications. Specifications are written in a high level, ARGUS-like language [Lis83]. Such specifications can be translated into versions that make the dependencies explicit; in connection with this we are investigating the use of data flow analysis techniques for an object preprocessor. In ISIS, the sequential ordering of two statements implies a data dependency between them. Control statements like if and while introduce data dependencies between the test and the body of the statement. If an operation does not depend on the value returned by a suboperation, the suboperation is labeled async. If operations $s_1, \ldots, s_k$ can be executed in parallel, we write $\texttt{cobegin } s_1 / s_2 / \cdots / s_k \texttt{ coend}$. Precedence dependencies between the branches of a $\texttt{cobegin } \ldots \texttt{ coend}$ statement are expressed as a partial ordering between the labels of the branches. Figure 2 shows the dependencies for the event scheduling operation using this construct; the translation for the above program is shown in Figure 3.

### 2.3. Conflicts Between Operations

Concurrency control algorithms identify conflicting operations and ensure that they are executed in a serializable order. A simple approach is to divide operations into classes and define conflicts between classes. For example, two-phase locking assumes that write operations on the same data item conflict with one another and with read operations. Read operations do not conflict with each other. This approach has been refined to achieve higher concurrency by defining lock classes depending on the semantics of higher level operations [Spec82][Ked83].

Similar to this refinement, the ISIS concurrency control scheme permits argument dependent operation conflicts to be defined. By default, operations on the same object are assumed to conflict, unless a conflict predicate indicates the absence of conflicts
between a pair of operations. Whether operations conflict generally depends on the semantics of the operations and thus has to be indicated as part of an object specification. Commutative operations do not conflict. Operations commute if their effect on an object is independent of the order in which they are executed. For example, B-tree insert operations are commutative: future operations return the same result regardless of the order of the inserts, although the internal structure of the tree might reflect this order. Let \( o_1 \) and \( o_2 \) be operations on the same object with arguments \( a_1, \ldots, a_k \) and \( b_1, \ldots, b_l \) respectively. Operations \( o_1 \) and \( o_2 \) commute if the predicate

\[
COMM_{o_1 \rightarrow o_2} \left( a_1, \ldots, a_k, b_1, \ldots, b_l \right)
\]

evaluates to true. In certain cases, it is possible to identify non-conflicting operations automatically in particular, if they access disjoint sets of data items. For the purpose of this paper we consider only conflict predicates based on commutativity. Our intent is to eventually generalize this approach to other types of conflict predicates.

The ISIS concurrency control mechanism evaluates conflict predicates at runtime to determine whether operations conflict. If they commute, they can be serialized in an order different from the serialization order of their parent operations. Normal locking schemes do not take advantage of this; our algorithm includes a mechanism to exploit this aspect to achieve greater concurrency.

### 2.4. Example

Figure 4 shows part of the specification of a resilient file system object with dependencies made explicit. The file system object maintains a list of free disk blocks and their size. The directory structure is maintained by a directory object. Files are created by invoking the create operation. To prevent creation of duplicate file names, the directory structure is first searched. Since the search is expected to fail in most cases, allocation of space for the new file is done in parallel with the search. If the file exists, the newly allocated space is deallocated and an error indication returned. Otherwise, the filename is inserted into the directory structure and the address of the file is returned. Note that the result of the create operation does not depend on the result of the insert,
the balance or the charge operation. This is indicated by the \texttt{async} keyword.

A file is accessed through read and write operations. We describe \texttt{write} in detail. For the sake of brevity, assume the data fit into the allocated space. The location of the file is retrieved from the directory object, and the data are copied into the file. The result returned does not depend on the directory update and the charges. The conflict predicate for \texttt{create(filename\textsubscript{c}, size\textsubscript{c})} and \texttt{write(filename\textsubscript{w}, buffer\textsubscript{w}, size\textsubscript{w})} expresses commutativity:

\[
\text{COMM}_{\text{create}, \text{write}}(\text{filename\textsubscript{c}, size\textsubscript{c}, filename\textsubscript{w}, buffer\textsubscript{w}, size\textsubscript{w}}) = (\text{filename\textsubscript{c}} \neq \text{filename\textsubscript{w}})
\]

3. Centralized Algorithm

The execution of operations can be represented by an \textit{execution log}: a multigraph, whose nodes correspond to operations. If an operation has no suboperations, it is represented by a leaf node in the execution log; these operations are called \textit{basic} operations. Basic operations access object data directly. \textit{Structured} operations are built from suboperations and are represented as an undirected, unordered tree. The root of the tree corresponds to the operation and its children correspond to the suboperations. Structured operations access data through their suboperations. For example, an \texttt{if then else} statement is represented as an \texttt{if}-node with three children for the condition, the then-part, and the else-part, respectively. Thus, undirected edges in the execution log reflect the nesting structure of an operation. Dependencies between operations appear as directed edges between the nodes. Figure 5 shows an execution log of the \texttt{create} operation. It is well known that an execution is serializable if its execution log is acyclic [Ber81][Ked83]. Data dependencies between operations are denoted by \text{\rightarrow}_{D}; precedence dependencies by \text{\rightarrow}_{P}. Nodes corresponding to operations which are being executed are marked \texttt{x}. An operation is called \textit{closed} if all of its suboperations are marked \texttt{x}, otherwise it is called \textit{open}.

The \texttt{ISIS} concurrency control algorithm maintains an execution log, which it uses to make serialization decisions. The centralized version of the algorithm is given in Figure 6a and 6b and described in detail below.

Inserting Precedence Edges

The serializer maintains an execution log of uncommitted operations. Once an operation is requested in an object, the coordinator first passes the tree representation of the operation together with its arguments to the serializer. The serializer evaluates conflict predicates to identify conflicts between operations in the log and nodes in the new tree. It then decides upon a serialization order for the highest distinct ancestors of the conflicting operations and introduces precedence edges to reflect this order, ensuring that there are no cycles of precedence edges. If this would require that an operation be serialized before a closed operation, the new one is aborted. The log is then used to schedule operations for execution.
3-resilient filesystem

created_by create_filesys; /* initializes new object instance */
operations create; write; read; delete;
calls directory;
create(filename, size) /* create a file if not already there, */
   string filename; integer size; /* allocate space, set up charges */
   
   { cobegin /* suboperation (1) */
      location1 = directory.search(filename);
      // /* suboperation (1.1) */
      // location2 = allocate(size);
      // /* suboperation (1.2) */
      coend;
      if(location1 != ERROR) /* suboperation (2) */
      { deallocate(location2);
         return(ERROR)
      }
      else /* suboperation (3) */
      {
         async cobegin { ins < bal } /* suboperation (4) */
            ins: directory.insert(filename, location2, size) /* suboperation (4.1) */
            // /* suboperation (4.2) */
            // charge(user, create) /* suboperation (4.3) */
            coend
          return(location2)
        }
    }

write(filename, buffer, size) /* write into a file */
   string filename; char buffer[]; integer size, location;
   
   { location = directory.search(filename);
     if(location == ERROR)
       return(ERROR);
     else
       cobegin
          copy(buffer, location, size);
          return(SUCCESS);
          // async directory.update(location, size);
          // charge(user, write);
       coend;
   }

end;

Figure 4: Specification of a resilient file system
Data dependencies also exist between every suboperation and its parent except between (4) and create, because (4) is labeled async.

Figure 5. Execution Log of Create

Wait-For Edges

To enforce a precedence dependency between two operations X and Y, suboperations in Y that conflict with suboperations in X must execute only after the suboperations in X are completed. To indicate this, the serializer introduces wait-for edges between conflicting operations: the operation at the head of the edge has to await completion of the operation at the tail. When the operation at the tail of a wait-for edge commits, the tail of the edge is promoted to its parent, because the operation at the head must now wait for the parent to commit. For example, in Figure 5, if A and B are two conflicting suboperations of (4.1) and (4.2) respectively, the serializer adds $A \rightarrow_w B$ to show that B must wait for A to complete. When A commits, the tail of the edge is promoted to (4.1).

When the operation at the tail of a precedence edge commits, the precedence dependency has been enforced. Therefore the tail of a wait-for edge is not promoted beyond the operation at the tail of the precedence edge that caused it - the wait-for edge is deleted instead. In the example above, the promotion continues until the tail reaches (4.1). When (4.1) completes the edge is removed. To determine when to delete wait-for edges, they are are labeled with the precedence edge that caused them.

The promotion of wait-for edges as described above corresponds directly to the lock inheritance in two-phase locking for nested transactions [Mos81]. In the example, greater concurrency would be achieved if B did not wait for the whole operation (4.1) to
complete. $B$ could be executed immediately after the completion of $A$ if it is known that $B$ does not conflict with any of the remaining suboperations of (4.1), and that (4.1) will not abort. In general, the tail of a wait-for edge can be promoted early if none of the remaining suboperations of the operation at the tail of the edge is going to conflict with the operation on the head of the edge, and if the operation at the tail will not abort. In the ISIS environment operations do not abort because of site failures, explicit aborts might appear in the specification. Thus we can decide whether the operation at the tail of a wait-for edge will abort. If early promotion causes a wait-for edge to be deleted the operation at its head can be executed before the conflicting operation actually commits.

Even if complete information about future operations is not available as described above, the early promotion scheme can still be used if conflicts and aborts are unlikely. Then, if a later conflict or abort occurs, the operation which was at the head of the wait-for edge must be aborted. This could lead to cascading aborts, but if edges are never promoted beyond top level operations, the "cascade" is always within some enclosing operation. Thus there is no need to abort a chain of operations, it is sufficient to abort the enclosing higher level operation.

**Executing Operations**

An operation is **executable** if it can be scheduled for execution; it is **completed** once its results are available, and it is **committed** once all of its suboperations are committed. A basic operation is executable if it does not have an incoming wait-for edge or a data dependency on any other operation. It can always be completed unless there are more failures than the object can tolerate. In this case the object becomes inaccessible and the operation blocks until the object recovers. A basic operation is committed once it is completed. A structured operation is executed by executing its suboperations, thus it is executable when at least one of its suboperations is executable. It is completed once all of the suboperations it has a data dependency on have been completed. Notice that the forward progress mechanism described in [Bir84a] makes it unnecessary to wait for operations on which there is no data dependency. Even if failures should occur, the mechanism guarantees that these operations will eventually be completed, thus guaranteeing that the entire operation is eventually finished in the traditional sense of the word. Structured operations are committed if all of their suboperations are committed.

**Data Dependency Edges**

All data dependencies are contained in the internal representation of an operation; the serializer includes them in the execution log. A data dependency edge on a lowest level operation is removed after the operation has been executed, data dependency edges on higher level operations are erased when they have no more data dependencies on any of their suboperations. Note that this means that if an operation $A$ does not have a data dependency on some of its suboperations, a data dependency on $A$ may be removed before all its suboperations commit. Thus an operation $B$ that required the result of $A$ can become executable as soon as the result of $A$ is available, and does not have to wait
for it to commit. This form of concurrency is not available in systems comparable to 
ISIS, although it is available in data-flow systems.

Deleting Precedence Edges

A precedence edge can always be deleted when the operation at its tail commits. The ISIS serialization algorithm takes advantage of commutativity between operations to delete a precedence edge earlier if possible. This increases concurrency because future operations are not constrained by this serialization order. While a pair of commutative operations is in progress, they have to be serialized to guarantee their atomicity with

Let $L$ be the representation of the execution log, and let $OP$ be the representation of the operation to be serialized. $OP$ is partially ordered by dependencies introduced in the definition of the operation.

For each operation $B$ in $OP$, bottom up, respecting the partial order, identify conflicting operations $A_i$ in $L$. For each pair $(A_i, B)$, let $X$ and $Y$ be the highest distinct ancestors of $A_i$ and $B$ respectively. Let $X = X_1, \cdots, X_k = A_i$ be the nodes on the path from $A_i$ to $X$ and let $Y = Y_1, \cdots, Y_l = B$ be the nodes on the path between $B$ and $Y$.

(1) if for all operations $X_m$ and $Y_n$, $COMM_{X_m, Y_n} = \text{FALSE}$:

- introduce $\rightarrow_p$ between $X$ and $Y$ such that the following conditions hold:
  - (a) $\rightarrow_p$ does not introduce a cycle of precedence edges
  - (b) If the new edge is $X \rightarrow_p Y$, then $B$ is open, else $A_i$ is open.
  - if this is not possible, reject $B$ and abort it.
- add edge $A_i \rightarrow_w B$ or $B \rightarrow_w A_i$, labeled $(X,Y)$ or $(Y,X)$, respectively, consistent with $\rightarrow_p$

- case (ii): There is already a dependency $X \rightarrow_p Y$ or $Y \rightarrow_p X$
  - add edge $A_i \rightarrow_w B$ or $B \rightarrow_w A_i$, consistent with $\rightarrow_p$
  - label $\rightarrow_w$ with $(X,Y)$ or $(Y,X)$, respectively.

(2) if there are operations $X_m$ and $Y_n$ such that $COMM_{X_m, Y_n} = \text{TRUE}$. Let $m$ and $n$ be maximal in lexicographical order.

- case (i): There is no dependency between $X$ and $Y$
  - same as case (i) above, except that the added edges are labeled $(X_m, Y_n)$ or $(Y_n, X_m)$

- case (ii): There is already a dependency $X \rightarrow_p Y$ or $Y \rightarrow_p X$
  - same as case (ii) above, except that the label is $(X_m, Y_n)$ or $(Y_n, X_m)$

**Figure 6a. The Centralized Edge Insertion Algorithm**
respect to other operations. Therefore they have to be treated the same way as non-commutative operations until one of them commits, and a precedence edge must be introduced between their highest common ancestors. Once either one of the operations has committed, their atomicity has been achieved. The serialization order between the pair is now irrelevant and the precedence edge introduced can be removed. To implement this, precedence edges introduced to serialize commutative operations are labeled with these operations. Labeled edges are removed once one of the operations in the label commits; unlabeled edges, i.e., edges introduced by non-commutative operations, are erased when the operation at the tail commits.

4. Decentralized Algorithm

To make resilient objects fault tolerant, the concurrency control algorithm must tolerate failures. In this section we introduce a fault tolerant decentralized version of the concurrency control algorithm described above.

4.1. Algorithm

A first approach to decentralizing the previous algorithm is to execute the centralized algorithm at all sites. To ensure that the same serialization order is obtained everywhere, it is necessary that new operations be added to each log in the same order. We achieve this by adding an information phase to the algorithm.

As before, the coordinator of an operation passes the internal representation to the local copy of the serializer; the serializer at that site then uses an atomic broadcast primitive [Gra78] to distribute the operation to other sites. The atomic broadcast primitive has the property that all operational sites receive the information or none does. Furthermore, if two sites initiate atomic broadcasts, the two pieces of information are received in the same order everywhere. After the atomic broadcast, each site inserts the new operation into its execution log following the rules described earlier. The resulting execution log is the same at all sites, since the order in which the operations are inserted is the same everywhere.

4.2. Optimizations

In the version described above, every operation is broadcast atomically to every site in the system. This imposes unnecessary overhead. Firstly, information about certain types of operations does not have to be broadcast immediately. For example, read operations can be done locally; they do not have to be serialized with respect to read operations at other sites. However, provision must be made to avoid the loss of read-serialization information in the event of failure. A solution to this problem is given in [Bir84b]; it piggybacks information about read operations on other messages and thus does not incur additional message overhead.

*If the cost of broadcasting large messages atomically is prohibitive, the atomic broadcast can be split into two parts. The bulk of data would be transmitted separately and only an “accept” message broadcast atomically.*
**Precedence Edges**

Let $X$ be a node corresponding to an operation that commits.
- Delete all edges $X\rightarrow p Y$ where $Y$ is any operation
- Delete all precedence edges labeled $(X, Y)$, where $Y$ is any operation

**Wait-for Edges**

Let $X$ be a node corresponding to an operation that commits and let $(A_1, B_1), \ldots, (A_m, B_m)$ be the labels on the $\rightarrow w$ edges originating in $X$.

For each edge $X\rightarrow w Y$ with label $(A_i, B_i)$ do
1. if $A_i = X$
   - delete $X\rightarrow w Y$
2. if $A_i$ is a predecessor of $X$
   - replace edge $X\rightarrow w Y$ by $\text{parent}(X)\rightarrow w Y$

For each edge $Y \rightarrow w Z$ with label $(A, B)$ do
   if $\text{PROMOTE}(Y, Z)$ is $\text{TRUE}$ /* PROMOTE described below */
   - if $Y = A$, delete $Y \rightarrow w Z$
   - if $Y \neq A$, replace $Y \rightarrow w Z$ by $\text{parent}(Y)\rightarrow w Z$

**Data Dependency Edges**

For each data dependency edge $X \rightarrow_D Y$.
1. if $X$ has no suboperations:
   - delete $X\rightarrow_D Y$ when $X$ commits
2. if $X$ has suboperations:
   - Let $Z_i, i = 1, \ldots, k$ be the suboperations on which $X$ has a data dependency.
   - delete $X\rightarrow_D Y$ once all $Z_i\rightarrow_D X$ are deleted.

**PROMOTE($X, Y$)**

$\text{PROMOTE}(X, Y)$ evaluates to $\text{TRUE}$, if $X \rightarrow w Y$ can be promoted to $\text{parent}(X)\rightarrow w Y$.

Let $S_1, \ldots, S_i$ be the unexecuted siblings of $X$.

In a conservative setting, the definition of $\text{PROMOTE}$ is as follows:

$\text{PROMOTE}(X, Y) = \text{TRUE}$, if for all $i$, $\text{COMM}_{S_i, Y} = \text{TRUE}$

In an optimistic setting, $\text{PROMOTE}(X, Y)$ is $\text{TRUE}$ if the above condition is "expected" to be true.

**Figure 6b. The Centralized Edge Deletion Algorithm**

Secondly, information about new operations need not be broadcast to sites executing operations whose serialization order cannot be affected by the new ones. The
algorithm uses the execution log to restrict the number of sites to which information is broadcast.

Specifically, the execution log is now viewed as consisting of several fragments: operations are in the same fragment if they are suboperations of the same operation, or if there are dependencies between them. Thus there are no edges between operations in different fragments. The objects accessed by the operations in a fragment are the objects of the fragment. The sites at which objects of a fragment reside are the sites of a fragment. A site can belong to more than one fragment if different fragments of the execution log contain operations on objects at that site. Whenever an operation in one fragment is serialized before a conflicting operation in another fragment, a precedence dependency between the two operations is introduced. As a result the two fragments have to be merged. When an operation commits, it is removed from the execution log; this might cause fragments to split. To accomplish this, the algorithm keeps track of fragments as they grow and shrink, and broadcasts information about operations only to sites in their fragments.

The algorithm maintains a list of the sites of a fragment at each of the sites. Initially, there are no fragments. A new operation can either be a suboperation of an existing operation, or it can be an independent operation. In the former case, the operation belongs to an existing fragment. If it contains suboperations on objects other than the ones of the existing fragment, the fragment is extended and the site-list for the fragment is updated accordingly. If the new operation is independent, it defines a fragment of its own. In either case, all the sites of the fragment are informed about the new operation and the new site-list, if any.

Once the serializer at a site receives a new operation, it checks for conflicts between it and operations known locally. If a conflict is detected between operations in different fragments, the algorithm enters a fragment merge phase. In this phase, a site that detects a conflict between two operations in different fragments merges the site-lists of the two fragments. The execution logs of both the fragments are known at this site; both the execution logs and the new site-list are broadcast to all sites of the new fragment. At the end of the fragment merge phase every site of the new fragment has complete information about the fragment.

In the third phase of the algorithm, the serialization phase, the serializers at each site in the fragment make serialization decisions according to the rules described in section 3.

Commitment of operations is broadcast to the sites of its fragment. The sites of the fragment remove the operation from their execution log. If the removal causes fragments to split, the fragments are redefined and site lists updated accordingly. Figure 7 shows the complete algorithm.
Let $o_{new}$ be the new operation to be serialized, and let $s$ be the site of the coordinator.

**Notification Phase**

At site $s$

- determine the fragment $c_{new}$ for $o_{new}$. Let $t$ be the set of sites of $c_{new}$.
- \texttt{atomic_bcast("INSERT, $o_{new}$, $c_{new}$", t)}

**Fragment Merge Phase**

At each site $s$, in $t$

if there is an operation known at $s$, that conflicts with $o_{new}$ and that is in a fragment $c \neq c_{new}$:

- assign to $t$ the site-list of $c$ merged with that of $c_{new}$
- assign to $e_{log}$ the execution log of $c$ merged with that of $c_{new}$
- \texttt{atomic_bcast("MERGE, $e_{log}$, $t$", $t$)}

**Serialization Phase**

At all sites

- serialize according to the rules of the centralized algorithm.

![Figure 7. The Decentralised Algorithm](image)

5. **Failures and Fault Tolerance**

Fault tolerance in ISIS is achieved through replication of object components at multiple sites. Each component is equally capable of executing an operation on the object. Site failures are detected through a low level failure detection protocol. The protocol starts whenever a failure is suspected, based on timeouts, and it terminates in two rounds of message exchange if no additional failures occur. If a site fails, the progress mechanism described in [Bir84a] ensures that operations supervised at the failed site are resubmitted by a new coordinator. The concurrency control algorithm recognizes these operations as having already been serialized, and uses the same serialization order. Thus failures do not compromise correct serialization.

6. **Conclusions**

In object oriented environments where objects can be specified as extensions of existing objects, it is important that the nesting of objects does not result in an implementation that is too inefficient to be viable. Consequently, it is desirable to attain as high a degree of concurrency as possible without introducing overhead. Object specifications contain semantic information that can be exploited to increase parallelism. In this paper, we have identified additional sources of concurrency: commutative operations can be executed in arbitrary order, in certain cases an operation can be executed before a conflicting operation commits, and data dependencies can be satisfied before an operation completes.
If speed of a resilient object is a critical factor, conflict predicates allow a user to provide additional information to increase concurrency within resilient objects. Although the conflict predicates considered in this paper are restricted to commutativity predicates, the concurrency control algorithm allows for general predicates; we expect to incorporate other types of conflict predicates in the near future.

We have given a specification technique that allows a user to express parallelism, dependencies and conflicts between operations. This information is used by the scheduling mechanism for concurrency control and optimization of message traffic. In particular, we have developed a fault-tolerant, distributed concurrency control algorithm which achieves a degree of concurrency comparable to optimistic concurrency control methods, but avoids their drawbacks. Furthermore, the algorithm does not incur the problem of lost read locks described in [Ber83] that can lead to non-serializable executions.

The implementation of a prototype of ISIS is well under way. The initial version contains a simplified version of the algorithm presented. We expect this prototype to provide measurements to assess the performance of the algorithm.

7. Acknowledgements

The authors would like to acknowledge the many contributions Dale Skeen, Amr El Abbadi, Wally Dietrich, Fred Schneider and Aliza Wechsler have made to the ISIS project.

8. References


