On the Correctness of Orphan Elimination Algorithms

(Extended Abstract)

Maurice Herlihy
Carnegie-Mellon Univ.
Dept. of Computer Sci.
Pittsburgh, PA.

Nancy Lynch
M.I.T.
Laboratory for Computer Sci.
Cambridge, MA.

Michael Merritt
AT&T Bell Laboratories
Murray Hill, N.J.

William Weihl
M.I.T.
Laboratory for Computer Sci.
Cambridge, MA.

1. Introduction

Nested transaction systems are being explored in a number of projects (e.g., see [6, 16, 13, 1]) as a means for organizing computations in a distributed system. Like ordinary transactions, nested transactions provide a simple mechanism for coping with concurrency and failures. In addition, nested transactions extend the usual notion of transactions [2, 12] to permit concurrency within a single action and to provide a greater degree of fault-tolerance, by isolating a transaction from a failure of one of its descendants.

In a distributed system, however, various factors, including node crashes and network delays, can result in *orphaned* computations: computations that are still running but whose results are no longer needed. As discussed in [7, 10], even if a system is designed to prevent orphans from permanently affecting shared data, orphans are still undesirable, for two reasons. First, they waste resources. Second, they may see inconsistent information. For example, a transaction might be reading data at two nodes, with some invariant relating the states of the data. If the transaction reads data at one of the nodes and then becomes an orphan, another transaction could change the data at both nodes before the orphan reads the data at the second node. This could happen, for example, because the first node learns that the transaction has aborted and releases its locks. While the inconsistencies seen by an orphan should not have any permanent effect on the shared data in the system, they can cause strange behavior if the orphan is interacting with the external world, and can also make programs difficult to design and debug.

Several algorithms have been designed to detect and eliminate orphans before they can see inconsistent information. In this paper we give formal descriptions and correctness proofs for the two orphan elimination algorithms in [7] and [10]. Our analysis covers only orphans resulting from aborts of actions that leave running descendants; we are currently working on modelling crashes and describing the algorithms that handle orphans that result from crashes. Our proofs are completely rigorous, yet quite simple. We show formally that the algorithms work in combination with any concurrency control protocol that ensures serializability of committed transactions, thus providing formal justification for the informal claims made by the algorithms' designers. Separating the orphan elimination algorithms from the concurrency control algorithms in this way contributes greatly to the simplicity of our results, and is in marked contrast to earlier work on similar problems (e.g., [4]).

The remainder of the paper is organized as follows. We begin in Section 2 with a brief description of *I/O automata*, which serve as the formal foundation for our work. Then, in Section 3, we review the model for nested transaction systems (including aborts) from [8]. The material in these two sections is largely abstracted from [8]; except for Section 3.5; the reader who is familiar with [8] is encouraged to skim these sections quickly.

In Section 4, we present some basic definitions and results that underlie the results to be presented in the rest of the paper. In Sections 5 — 8, we present a series of different systems. Each involves the same transactions and generic objects as the generic system, and each ensures orphan elimination by using a modified controller. Two are "abstract algorithms" that use global information and are easy to prove correct directly. The others, which model the algorithms from [7] and [10], use local information, and are verified by showing that they simulate the abstract algorithms. Because of lack of space in the proceedings, some details are omitted. A complete version of the paper can be obtained from the authors.

2. Basic Model

We use the I/O automaton model [8, 9], a simple model for concurrent systems, as the formal foundation for our work. This model consists of (possibly infinite-state) nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. In this paper, we only prove properties of finite behavior, so we only require a simple special case of the general model. In this section, we give a concise review of the relevant definitions.

2.1. I/O Automata

An I/O automaton A has components *states*(A), *start*(A), *out*(A), *in*(A), and *steps*(A). Here, *states*(A) is a set of states, of which a subset, *start*(A), is designated as the set of start states. The next two components are disjoint sets: *out*(A) is the set of *output operations*, and *in*(A) is the set of *input operations*. The union of these two sets is the set of *operations* of the automaton. Finally, *steps*(A) is the transition relation of A, which is a set of triples of the form (s',π,s), where s' and s are states, and π is an operation. Such a triple means that in state s', the automaton can atomically do operation π and change to state s. An element of the transition relation is called a *step* of A. If (s',π,s) is a step of A, we say that π is *enabled* in s'.

The output operations model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. We require the following *input condition*, which says that an I/O automaton must be prepared to receive any input operation at any time: For each input operation π and each state s', there exist a state s and a step (s',π,s).

An *execution* of A is a finite alternating sequence $s_0,\pi_1,s_1,\pi_2,\ldots$ of states and operations of A, ending with a state. Furthermore, s_0 is in start(A), and each triple (s',π,s) that occurs as a consecutive subsequence is a step of A. From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of operations only. Because transitions to different states may have the same operation, different executions may have the same schedule.

If S is any set of schedules (or property of schedules), then A is said to *preserve* S provided that the following holds. If $\alpha = \alpha'\pi$ is any schedule of A, where π is an output operation, and α' is in S, then α is in S. That is, the automaton is not the first to violate the property described by S.

The work of the first author was supported by a grant from the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520. The work of the second author was supported in part by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, the Office of Naval Research under Contract N00014-85-K-0168, and by the Office of Army Research under Contract DAAG29-84-K-0058. The work of the fourth author was supported in part by an IBM Faculty Development Award, the National Science Foundation under grant DCR-85-100014, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

2.2. Composition of Automata

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata, also. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton.

A set of I/O automata may be composed to create a *system* S, if the sets of output operations for the automata are disjoint. (Thus, every output operation in S will be triggered by exactly one component.) The system S is itself an I/O automaton. A state of the composed automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The set of *operations* of S, $ops(S)$, is exactly the union of the sets of operations of the component automata. The set of *output operations* of S, $out(S)$, is likewise the union of the sets of output operations of the component automata. Finally, the set of *input operations* of S, $in(S)$, is $ops(S) - out(S)$, the set of operations of S that are not output operations of S. The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.

The triple (s',π,s) is in the transition relation of S if and only if for each component automaton $\mathcal{A}$, one of the following two conditions holds. Either π is an operation of $\mathcal{A}$, and the projection of the step onto $\mathcal{A}$ is a step of $\mathcal{A}$, or else π is not an operation of $\mathcal{A}$, and the states corresponding to $\mathcal{A}$ in the two tuples s' and s are identical. During an operation π of S, each of the components that has operation π carries out the operation, while the remainder stay in the same state.

If α is a sequence of operations of a system S with component $\mathcal{A}$, then we denote by $\alpha|\mathcal{A}$ the subsequence of α containing all the operations of $\mathcal{A}$. Clearly, if α is a schedule of S, $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$.

3. Generic Systems

In this section, we define "generic systems", which consist of transactions, generic objects, and a generic controller. They are a generalization of the "weak concurrent systems" of [8]. Transactions and generic objects describe user programs and data, respectively. The generic controller controls communication between the other components, and thereby defines the allowable orders in which the transactions may take steps. All three types of system components are modelled as I/O automata.

We begin by defining a structure that describes the nesting of transactions. Namely, a *system type* is a four-tuple $(\mathcal{T},\text{parent},O,V)$, where $\mathcal{T}$, the set of transaction names, is organized into a tree by the mapping parent: $\mathcal{T} \rightarrow \mathcal{T}$, with T_0 as the root. The leaves of this tree are called *accesses*. The set O denotes the set of objects; formally, O is a partition of the set of accesses, where each element of the partition contains the accesses to a particular object. The set V is a set of *values*, to be used as return values of transactions.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of a "mythical" transaction, T_0, the root of the transaction tree. It is convenient to introduce the root transaction to model the environment in which the rest of the transaction system runs. Transaction T_0 has operations that describe the invocation and return of the classical transactions.

The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. The only transactions that actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". The partition O simply identifies those transactions that access the same object.

A generic system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each *internal* (i.e. non-leaf, non-access) node of the transaction tree, a generic object automaton for each element of O, and a generic controller. These automata are described below. (If X is a generic object associated with an element X of the partition O, and T is an access in X, we write $T \in accesses(X)$ and say that "T is an access to X".)

For the rest of this paper, we fix a particular system type $(\mathcal{T},\text{parent},O,V)$.

3.1. Transactions

A non-access transaction T is modelled as an I/O automaton, with the following operations:

Input operations:
- CREATE(T)
- COMMIT(T',v), for T' ∈ children(T) and v ∈ V
- ABORT(T'), for T' ∈ children(T)

Output operations:
- REQUEST_CREATE(T'), for T' ∈ children(T)
- REQUEST_COMMIT(T,v), for v ∈ V

The CREATE input operation "wakes up" the transaction. The REQUEST_CREATE output operation is a request by T to create a particular child transaction. The COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The ABORT input operation reports to T the unsuccessful completion of one of its children. We call COMMIT(T',v), for any v, and ABORT(T') *return* operations for transaction T'. The REQUEST_COMMIT operation is an announcement by T that it has finished its work.

We leave the executions of particular transaction automata largely unspecified; the choice of which children to create, and what value to return, will depend on the particular implementation. However, it is convenient to assume that schedules of transaction automata obey certain simple syntactic constraints (described in the full paper).

3.2. Generic Objects

In this section, we define the aspects of generic objects that are relevant to our analysis of orphan algorithms. It turns out that the details of how synchronization and recovery are implemented by a generic object are largely irrelevant. Indeed, this is one of the important contributions of this paper: we are able to state correctness conditions for and verify orphan elimination algorithms in a way that is completely independent of the concurrency control and recovery method used.

A generic object X is modelled as an I/O automaton, with the following operations:

Input Operations:
- CREATE(T), T an access to X
- INFORM_COMMIT_AT(X)OF(T)
- INFORM_ABORT_AT(X)OF(T)

Output Operations:
- REQUEST_COMMIT(T,v), T an access to X

The CREATE input operation starts an access transaction at the object. (Thus, it corresponds to the invocation of an instance of one of the object's "operations".) Similarly, the REQUEST_COMMIT output indicates that an access transaction has finished its work, and includes a value recording the results. The INFORM_COMMIT and INFORM_ABORT input operations tell X that some transaction (not necessarily an access to X) has committed or aborted, respectively.

As for transaction automata, we leave the executions of particular generic objects largely unspecified. However, we do assume, as for transactions, that schedules of generic objects obey certain syntactic constraints (again, described in the full paper).

3.3. Generic Controller

The third kind of component in a generic system is the generic controller. The generic controller is also modelled as an automaton. The transactions and generic objects have been specified to be any

The generic controller has seven operations:

Input Operations:

REQUEST_CREATE(T),
REQUEST_COMMIT(T,v).

Output Operations:

CREATE(T),
COMMIT(T,v),
ABORT(T),
INFORM_COMMIT_AT(X)OF(T),
INFORM_ABORT_AT(X)OF(T).

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and object automata, and correspondingly for the output operations.

Each state s of the generic controller consists of five sets: create_requested(s), created(s), commit_requested(s), committed(s), and aborted(s). The set commit_requested(s) is a set of (transaction,value) pairs, and the others are sets of transactions. The initial state of the generic controller is denoted by s_0. All of the components of s_0 are empty except for create_requested, which is $\{T_0\}$. For a state s, we define returned(s) = committed(s) ∪ aborted(s).

The transition relation for the generic controller consists of exactly those triples (s',π,s) satisfying the preconditions and postconditions below, where π is the indicated operation. For brevity, we include in the postconditions only those conditions on the state s that may change with the operation. If a component of s is not mentioned in the postcondition the component is taken to be the same in s as in s'.

- REQUEST_CREATE(T)
  Postcondition:
  create_requested(s) = create_requested(s') ∪ {T}
- REQUEST_COMMIT(T,v)
  Postcondition:
  commit_requested(s) = commit_requested(s') ∪ {(T,v)}
- CREATE(T)
  Precondition:
  T ∈ create_requested(s') — created(s')
  Postcondition:
  created(s) = created(s') ∪ {T}
- COMMIT(T,v)
  Precondition:
  (T,v) ∈ commit_requested(s')
  T ∉ returned(s')
  children(T) ∩ create_requested(s') ⊆ returned(s')
  Postcondition:
  committed(s) = committed(s') ∪ {T}
- ABORT(T)
  Precondition:
  T ∈ create_requested(s') — returned(s')
  Postcondition:
  aborted(s) = aborted(s') ∪ {T}
- INFORM_COMMIT_AT(X)OF(T):
  Precondition:
  T ∈ committed(s')
- INFORM_ABORT_AT(X)OF(T):
  Precondition:
  T ∈ aborted(s')

The controller simply records its input operations in the appropriate components of the state. Similarly, the postconditions for COMMIT and ABORT record that the operation has occurred. Once the creation of a transaction has been requested, the controller can create it by producing a CREATE operation. INFORM_COMMIT and INFORM_ABORT operations can be generated at any time after the corresponding COMMIT and ABORT operations have occurred.

The precondition for the COMMIT operation ensures that a transaction only commits if it has requested to do so, and has not already returned. In addition, the actual COMMIT operation must be delayed until all children requested by the committing transaction have returned. Notice that there are few constraints on when a transaction can be aborted. For example, a transaction can be aborted while some of its descendants are still running.

In the material that follows we will rely on some simple invariants relating schedules of the generic controller to the states that result from applying them to the initial state. For example, if α results in state s, then T is in aborted(s) exactly if α contains an ABORT(T) operation. For brevity in this paper, however, we omit the detailed statements of these invariants.

3.4. Generic Systems

The composition of transactions with generic objects and the generic controller is called a *generic system* (of the given system type). The non-access transactions and the generic objects are called the system *primitives*. The schedules of a generic system are called *generic schedules*, and the operations are called *generic operations*. For any generic operation π, we define *location*(π) to be the primitive at which π occurs. (Each operation occurs both at a primitive and at the generic controller; no operation, however, occurs at more than one primitive.)

3.5. Correctness

In much of the database literature on transactions, serializability is taken as the definition of correctness. To deal with nested transactions, and to handle aborts, the usual notion of serializability must be generalized. This is done in [8] as follows.

A generic system is correct if every schedule of the generic system "looks like" a serial schedule to the transactions. The permissible serial schedules are defined by another kind of system, called a "serial system". Serial systems are similar to generic systems in that they are composed of transactions, a serial controller, and objects. The transactions are identical to those in generic systems. The serial controller, however, differs from the generic controller in two respects. First, the serial controller permits only one child of a transaction to run at a time. Thus, sibling transactions execute sequentially at every level in the transaction tree, so that transactions are run in a depth-first traversal of the tree. Second, the serial controller aborts a transaction only if it has not yet been created, and creates a transaction only if it has not been aborted. In other words, aborted transactions never take any steps in a serial schedule.

Objects in a serial system are simpler than generic objects. Since the serial controller guarantees that siblings execute sequentially, and that aborted transactions never take any steps, serial objects do not have to deal with concurrency or with failures. The serial objects serve as a specification of how objects should behave in the absence of concurrency and failures. (The serial objects in [8] serve the same purpose as the "serial specifications" in [17].)

Many possible notions of correctness can be defined. We consider two here. The first is quite simple: it requires that every schedule look like a serial schedule to every transaction. More precisely, if α is a generic schedule and T is a non-access transaction, we say that α *is serially correct at T* if there exists a serial schedule β such that $\beta|T = \alpha|T$. In other words, T sees the same thing in α that it could see in some serial schedule. We say that α is *serially correct* if it is serially correct for all non-access transactions. We also say that a system is serially correct if every schedule of the system is serially correct.

Requiring every transaction to see a serial view is a strong requirement. Without orphan elimination, in fact, systems may not meet this requirement. Instead, they provide a slightly weaker notion of correctness, namely that non-orphan transactions see serial views. More precisely, if α is a sequence of generic operations and T is a transaction, we say that T is an *orphan* in α if ABORT(T')

occurs in α for some ancestor T' of T. Systems without orphan elimination ensure that each schedule is serially correct for all non-orphan transactions; orphan transactions, however, can see arbitrary views.

In [8], an example is given for a particular kind of generic system which guarantees serial correctness for non-orphan transactions. In the system of that paper, each generic object is the composition of a "resilient object" and a corresponding "lock manager". The resilient object handles recovery processing. The lock manager implements an exclusive locking protocol based on that of Moss [11]. The combination can be encapsulated in a generic object, which handles both concurrency control and recovery. In this paper, we call the combination of a resilient object and a lock manager a *locking object*, and we call a generic system built using locking objects a *locking system*. We call schedules of a locking system *locking schedules*. The following theorem expresses the correctness guarantee proved in [8] for locking systems.

Theorem 1: Let α be a locking schedule and let T be a non-access transaction that is not an orphan in α. Then α is serially correct at T.

The orphan elimination algorithms of this paper ensure that schedules are serially correct for all non-access transactions, both orphans and non-orphans. To ensure this, the orphan elimination algorithms rely on the generic objects to ensure serial correctness for non-orphans; in fact, the algorithms work with any generic objects that ensure serial correctness for non-orphans. Thus, the orphan elimination algorithms and the concurrency control algorithms are independent.

Theorem 1 implies that locking objects can be used with the orphan elimination algorithms to yield serial correctness. There are also many other examples of suitable generic objects. For example, it is shown in [3] that objects that use read-write locking instead of exclusive locking ensure serial correctness for non-orphans. We are also currently working on generalizing the results in [17] to nested transaction systems. This will permit us to show that many other kinds of objects ensure serial correctness for non-orphans, including objects that use timestamps for concurrency control [14], and objects that use more general approaches to locking [5, 15, 17]. The results in this paper indicate that the orphan elimination algorithms analyzed here can be combined with any of these objects.

4. Information Flow

The orphan elimination algorithms analyzed in this paper use quite different techniques to detect and eliminate orphans. However, the fundamental underlying structure is quite similar. In this section we define a notion of a "dependency relation" that models the information flow among operations. These definitions allow us to analyze both orphan elimination algorithms in a simple and straightforward manner.

For a sequence α of generic operations, define the relation *directly-affects(α)* to be the relation containing the pairs (ϕ, π) of operation instances[5] such that ϕ occurs before π in α, and at least one of the following holds:

- location(ϕ) = location(π), and π is an output operation of the primitive
- ϕ = REQUEST_CREATE(T) and π = CREATE(T)
- ϕ = REQUEST_COMMIT(T,v) and π = COMMIT(T,v)
- ϕ is a return operation for a child of T and π = COMMIT(T,v)
- ϕ = REQUEST_CREATE(T) and π = ABORT(T)
- ϕ = COMMIT(T,v) and π = INFORM_COMMIT_AT(X)OF(T)
- ϕ = ABORT(T) and π = INFORM_ABORT_AT(X)OF(T)

Define the relation *affects(α)* to be the transitive closure of directly-affects(α). If the pair (ϕ,π) is in the relation directly-affects(α), we say that ϕ *directly-affects* π in α. Similarly, if (ϕ,π) is in the relation affects(α), we say that ϕ *affects* π in α.

The idea is that ϕ directly-affects π if it is possible for the precondition for π to require ϕ to have occurred earlier. We want to make as few assumptions about the particular primitives used in a system. Thus, we make the "safe" choice of assuming a dependency whenever one could occur. (For operations involving different primitives, the transition relation of the generic controller tells us exactly which dependencies exist.) Fortunately, the orphan elimination algorithms described later in this paper are independent of the particular primitives used in a system, and do not rely on more information about them.

If α is a sequence of generic operations and β is a subsequence of α, we say that β *is closed in* α if, whenever β contains an operation instance π in α, it also contains any ϕ that affects π in α.

The following lemma states that affects(α) contains all dependencies that are relevant to the execution of a generic system.

Lemma 2: If α is a generic schedule, then any closed subsequence of α is also a generic schedule.

In other words, if π is not affected by ϕ in some schedule α, then π cannot "know" that ϕ occurred, since π could also have occurred in a different schedule in which ϕ did not occur.

5. Filtered Systems

One way of ensuring that operations of a transaction T are never affected by the abort of an ancestor of T is to add preconditions to the operations of the generic controller to permit operations of T to occur only if they would not be affected in this way. It turns out, however, that this approach checks for orphans much more frequently than necessary. In this section we define another kind of system, called a "filtered system", that checks for orphans only when access transactions commit. We then show that this is sufficient to ensure that transactions are never affected by the aborts of ancestors.

Filtered systems consist of transactions, generic objects, and a "filtered controller". The filtered controller is obtained by slightly modifying the generic controller; it "filters" commits of access transactions so that any non-access transaction, orphan or not, sees a view it could see as a non-orphan in the generic system.

5.1. The Filtered Controller

The filtered controller has the same seven operations as the generic controller. Each state s of the filtered controller consists of six components. The first five are the same as for the generic controller. The sixth, history(s), is a sequence of generic operations. The initial state of the filtered controller is denoted by s_0. As in the generic controller, all sets are empty in s_0 except for create_requested, which is $\{T_0\}$. History(s_0) is the empty sequence. As before, we define returned(s) = committed(s) $\cup$ aborted(s).

The transition relations for all operations except COMMIT(T,v), where T is an access, are defined as for the generic controller, except that each operation π has an additional postcondition of the form history(s) = history(s')π. In other words, the history component of the state simply records the sequence of operations that have occurred. The transition relation for the COMMIT(T,v) operation, where T is an access, is defined as follows.

- COMMIT(T,v), T an access
  Precondition:
  (T,v) $\in$ commit_requested(s')
  T $\notin$ returned(s')

[5] Formally, an *operation instance* is a pair (i,π), where i is a positive integer and π is an operation. An operation instance (i,π) is said to *occur* in α if the i-th element of α is π. The distinction is exactly that of a symbol (operation) and the occurrence of the symbol in a string (operation instance).

if T' is an ancestor of T,
then ABORT(T') does not affect COMMIT(T,v)
in history(s')COMMIT(T,v)
Postcondition:
committed(s) = committed(s') ∪ {T}
history(s) = history(s')COMMIT(T,v)

5.2. Filtered Systems

A *filtered system* is the composition of transactions, generic objects and the filtered controller. Schedules of a filtered system are called *filtered schedules*.

Lemma 3: Every filtered schedule is a generic schedule.

As described above, the filtered controller performs an explicit test to ensure that the commit of an access is not affected by the abort of any ancestor. The following key lemma shows that this test actually guarantees more: that a similar property holds for all operations occurring at non-access transactions.

Lemma 4: Let α be a filtered schedule, and let T be a non-access transaction. Let ρ be an operation in α, such that location(ρ) = T. Then there is no ABORT(T') operation that affects ρ in α, for any ancestor T' of T.

5.3. Simulation of Generic Systems by Filtered Systems

The following theorem is the key result of the paper. It shows that filtered systems ensure that every transaction gets a view it could get when it is not an orphan. (Formally, a transaction T's "view" in a schedule α is its local schedule, $\alpha|T$.) In other words, an orphan cannot discover that it is an orphan, since the view it sees is consistent with it not being an orphan. This is the basic correctness property for the orphan elimination algorithms.

Theorem 5: Let α be a filtered schedule and let T be a non-access transaction. Then there exists a generic schedule β such that T is not an orphan in β and $\beta|T = \alpha|T$.

Proof: Let β be the subsequence of α containing all operations π such that location(π) = T, and all other operations ϕ that affect, in α, some operation whose location is T. Since affects(α) is a transitive relation, β is closed in α. By Lemma 2, β is a generic schedule. It suffices to show that there is no ancestor T' of T for which ABORT(T') occurs in β. Suppose not; i.e., there exists an ancestor T' of T for which ABORT(T') occurs in β. Then by the construction of β, α contains an operation π of T such that ABORT(T') affects π in α. By Lemma 4, this is impossible. □

As discussed earlier, we can combine Theorem 5 with Theorem 1 to obtain an important corollary. Define a *filtered locking system* to be a filtered system whose generic objects are locking objects; its schedules are called *filtered locking schedules*.

Corollary 6: Any filtered locking system is serially correct.

Proof: Let α be a filtered locking schedule and let T be a non-access transaction. Theorem 5 yields a locking schedule γ such that T is not an orphan in γ and $\gamma|T = \alpha|T$. Theorem 1 then yields a serial schedule β with $\beta|T = \gamma|T$; this is equal to $\alpha|T$, as needed. □

A similar corollary can be obtained for any generic system whose transactions and objects ensure serial correctness for non-orphans.

It is not necessary to filter operations other than commits of accesses because the communication patterns among the primitives in a system are restricted. The execution of a transaction primitive T can be affected by an ancestor only through the CREATE(T) operation, or through communication via shared objects. As long as T does not receive replies (commits) from any objects that "know" that its ancestor has aborted, T cannot observe a state that depends on the abort.

6. Argus Systems

In this section we analyze the orphan elimination algorithm used in the Argus system [6, 7]. We describe the algorithm by defining an *Argus controller* that describes in formal terms the algorithm discussed in [7]. We then define Argus systems, which are composed of transactions, generic objects, and an Argus controller, and show that Argus systems "simulate" filtered systems. In other words, a schedule of an Argus system looks like a schedule of a filtered system to each non-access transaction; if the filtered system is serially correct, then so is the corresponding Argus system.

6.1. The Argus Controller

The filtered controller uses global knowledge of the entire history of operations to filter the commits of access transactions. This kind of global knowledge is not practical in a distributed system. Thus, the Argus algorithm makes use of local knowledge about the aborts that have occurred. To ensure that the commit of an access is not affected by the abort of an ancestor, the Argus algorithm keeps track of the aborts "known" by each operation that occurs, and propagates this knowledge from an operation to any later operations that it affects.

The Argus controller has the same seven operations as the generic controller. Each state s of the Argus controller consists of six components. The first five are the same as for the generic controller (i.e., create_requested(s), created(s), commit_requested(s), committed(s), and aborted(s)). The sixth, done(s), is a mapping from operations to sets of transactions. This mapping records the transactions whose aborts affect each operation, as the execution proceeds. (The set done(s)(π) may actually include more transactions than those whose aborts affect π in the execution. By adding more aborted transactions to this set, an implementation would effectively restrict the behavior of orphans further than is strictly necessary to ensure the correctness conditions. Thus, we might say that done(s)(π) contains those transactions that π "knows" are aborted.)

As before, the initial state is denoted by s_0, and all sets are initially empty in s_0 except for create_requested, which is $\{T_0\}$. The function done(s_0) maps each operation to the empty set.

For brevity, we describe only how the transition relation for the Argus controller differs from that of the generic controller. First, each operation π contains the following additional postcondition: for every operation ϕ (including π), done(s')(ϕ) ⊆ done(s)(ϕ). In other words, information "known" by an operation is not lost over time. Second, each operation π has an additional postcondition of the following form: for every ϕ such that ϕ directly-affects π, done(s')(ϕ) ⊆ done(s)(π). In other words, if ϕ is known by π to have occurred, then all aborts known by ϕ are known by π. Thus, for example, the CREATE(T) operation has an additional postcondition that done(s')(REQUEST_CREATE(T)) ⊆ done(s)(CREATE(T)). Finally, we modify the COMMIT operation for accesses and the ABORT operation as follows:

- COMMIT(T,v), T an access
  Precondition:
  (T,v) ∈ commit_requested(s')
  T ∉ returned(s')
  there is no ancestor of T in
  done(s')(REQUEST_COMMIT(T,v))
  Postcondition:
  committed(s) = committed(s') ∪ {T}
  done(s')(REQUEST_COMMIT(T,v)) ⊆
  done(s)(COMMIT(T,v))
- ABORT(T)
  Precondition:
  T ∈ create_requested(s') — returned(s')
  Postcondition:
  aborted(s) = aborted(s') ∪ {T}
  done(s')(REQUEST_CREATE(T)) ∪ {T} ⊆
  done(s)(ABORT(T))

6.2. Argus Systems

An *Argus system* is the composition of transactions, generic objects, and the Argus controller. Schedules of the Argus system are called *Argus schedules*.

Lemma 7: Every Argus schedule is a generic schedule.

The next lemma shows that the postconditions on the operations are enough to ensure that done(s)(ϕ) contains T whenever ABORT(T) affects an instance of ϕ.

Lemma 8: Let α be an Argus schedule such that an instance of ABORT(T) affects an instance of operation ϕ in α, and let s be a state of the Argus controller after α. Then T $\in$ done(s)(ϕ).

6.3. Simulation of Generic Systems by Argus Systems

The following lemma shows that the information in done(s), combined with the precondition on commits of accesses, is enough to ensure that Argus systems simulate filtered systems.

Lemma 9: Every Argus schedule is a filtered schedule.

Given this lemma, it is a simple exercise to show the analogs for the Argus controller of Theorem 5 and Corollary 6.

7. Strictly Filtered Systems

Because of lack of space in the proceedings, we present only a sketch of the results of this section. The idea is to define a "strictly filtered controller", which allows an access to commit only if no ancester has aborted. It is simple to show that this controller simulates the filtered controller. However, it uses global information about the aborts that have occurred. In the next section, we show that the algorithm from [10], which uses local information, simulates the strictly filtered controller.

8. Clock Systems

Because of lack of space, we can only sketch the results here. We model the **algorithm from** [10] by a "clock controller". The clock controller maintains a quiesce time for each access transaction and a release time for every transaction. An access transaction is allowed to commit only if its quiesce time has not passed. Release times are chosen so that once a transaction's release time is reached, all its descendant accesses have quiesced. A transaction is allowed to abort only if its release time has passed. This ensures that, after a transaction aborts, none of its descendant accesses will commit. Operations are provided to model the passage of time, and to permit quiesce and release times to be adjusted. We then show that the clock controller simulates the strictly filtered controller.

9. Conclusions

We have defined correctness properties for orphan elimination algorithms, and have presented precise descriptions and proofs for two algorithms from [7] and [10]. Our proofs are quite simple, and show that the systems exhibit a substantial degree of modularity: the orphan elimination algorithms can be used in combination with any concurrency control protocol that ensures correctness for non-orphans. The simplicity of our proofs is a direct result of this modularity, and is in sharp contrast to earlier work [4], in which the orphan elimination algorithm and the concurrency control protocol were not cleanly separated.

In this paper we have analyzed only orphans that result from aborts of transactions. We are currently studying orphans that result from crashes. The algorithms for detecting and eliminating such orphans described in [7, 10] are quite interesting, but also more complicated than the algorithms for handling aborts. We would like to find a similar separation of concerns for the crash-orphan algorithms, showing, for example, that the crash-orphan algorithms are independent of the concurrency control protocol and the abort-orphan algorithm used in the system. Whether this will be possible is still unknown.

References

[1] Allchin, J. E.
An architecture for reliable decentralized systems.
PhD thesis, Georgia Institute of Technology, September, 1983.
Available as Technical Report GIT-ICS-83/23.

[2] Bernstein, P. A., and Goodman, N.
Concurrency control in distributed database systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[3] Fekete, A., Lynch, N. A., Merritt, M. and Weihl, W. E.
Nested transactions and read/write locking.
In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 97-111. 1987.

[4] Goree, J. A.
Internal consistency of a distributed transaction system with orphan detection.
Master's thesis, MIT, January, 1983.
Available as MIT/LCS/TR-286.

[5] Korth, H.
Locking primitives in a database system.
JACM 30(1), January, 1983.

[6] Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[7] Liskov, B., Scheifler, R., Walker, E. F., and Weihl, W.
Orphan Detection.
In *this proceedings*. IEEE, July, 1987.

[8] Lynch, N. A., and Merritt, M.
Introduction to the theory of nested transactions.
Technical Report MIT-LCS-TR-367, Massachusetts Institute of Technology, 1986.
Appeared in Proceedings of 1986 International Conference on Database Theory.

[9] Tuttle, M. and Lynch, N. A.
Hierarchical correctness proofs for distributed algorithms.
Submitted for publication.

[10] McKendry, M., and Herlihy, M.
Time-driven orphan elimination.
In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 42-48. IEEE, January, 1986.

[11] Moss, J. E. B.
Nested transactions: an approach to reliable distributed computing.
PhD thesis, Massachusetts Institute of Technology, 1981.
Available as Technical Report MIT/LCS/TR-260.

[12] Papadimitriou, C.H.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[13] Pu, C., and Noe, J. D.
Nested transactions for general objects: the Eden implementation.
Technical Report TR-85-12-03, University of Washington Department of Computer Science, December, 1985.

[14] Reed, D.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[15] Schwarz, P., and Spector, A. Z.
Synchronizing shared abstract types.
ACM Transactions on Computer Systems 2(3), August, 1984.

[16] Spector, A. Z., et al.
Support for distributed transactions in the TABS prototype.
In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*. IEEE, October, 1984.

[17] Weihl, W. E.
Specification and implementation of atomic data types.
PhD thesis, Massachusetts Institute of Technology, 1984.
Available as Technical Report MIT/LCS/TR-314.