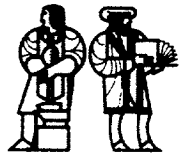
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-329

ON THE CORRECTNESS OF ORPHAN ELIMINATION ALGORITHMS

MAURICE HERLIHY
NANCY LYNCH
MICHAEL MERRITT
WILLIAM WEIHL

MAY 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

On the Correctness of Orphan Elimination Algorithms

Maurice Herlihy[1]

Nancy Lynch[2]

Michael Merritt[3]

William Weihl[4]

Abstract

In a distributed system, node crashes and network delays can result in *orphaned* computations: computations that are still running but whose results are no longer needed. Several algorithms have been proposed to detect and eliminate such computations before they can see inconsistent states of the shared, concurrently accessed data. In this paper we analyze two orphan elimination algorithms that have been proposed for nested transaction systems. We describe the algorithms formally, and present complete detailed proofs of correctness. Our proofs are remarkably simple, and show that the fundamental concepts underlying the two algorithms are quite similar. In addition, we show formally that the algorithms can be used in combination with any correct concurrency control technique, thus providing formal justification for the informal claims made by the algorithms' designers. Our results are a significant advance over earlier work in the area, in which it was extremely difficult to state and prove comparable results.

Keywords: orphans, distributed databases, transactions, serializability, concurrency control, recovery.

---

[1]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. Address: CMU Department of Computer Science, Pittsburgh, PA

[2]Supported 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. Address: MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA

[3]Address: AT&T Bell Laboratories, Murray Hill, NJ

[4]Supported 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. Address: MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA

1. Introduction

Nested transaction systems are being explored in a number of projects (e.g., see [6, 19, 16, 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, 15] 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. For example, in the Argus system [6], the node making a remote request may give up because a network partition or some other problem makes it impossible to communicate with the other node. This may leave a process running at the called node; this process is an orphan. The orphan runs as a descendant of the transaction that made the call. Since the caller gives up by aborting the transaction that made the call, the orphan must not have any permanent effects on the observed state of the shared data.

As discussed in [7, 11], 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: they use processor cycles, and may also hold locks, causing other computations to be delayed. 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 proposed to prevent orphans from seeing inconsistent information. Early work in the area includes Nelson's thesis [14], which describes algorithms for detecting and eliminating orphans that arise because of crashes. Nelson's work did not assume an underlying transaction mechanism, so that it was difficult to assign a simple semantics to abandoned computations. Recent work [20, 7, 11] has studied orphans in the context of a nested transaction system, in which an abandoned computation can be *aborted*, preventing it from having any effect on the state of the system. The goal of the algorithms in [20, 7, 11] is 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 [11]. The algorithm in [7] is currently in use in the Argus system. 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. In addition, both the presentations and the proofs follow the intuitions that the designers use in describing the algorithms.

Our proofs clarify the fundamental concepts underlying the algorithms. We define a single general correctness condition for algorithms that eliminate orphans that result from aborts, and prove that both algorithms ensure this condition. While the algorithms seem quite different, our proofs show that the underlying ideas are in fact very similar.

The designers of the algorithms have claimed that the algorithms work in combination with any concurrency control protocol that ensures serializability of committed transactions. Our correctness condition relates the behavior of a system containing an orphan elimination algorithm to a system with no orphan elimination: the system with orphan elimination must "simulate" the system without orphan elimination, in the sense that each transaction in the system with orphan elimination must see a view of the system it could see in an execution of the other system in which it was not an orphan. (A transaction's "view of the system" is its sequence of interactions with the system, including the results of operations invoked by the transaction.) Thus, if the concurrency control protocol ensures that non-orphans see consistent views, our correctness condition implies that the orphan elimination algorithm will guarantee that all transactions see consistent views. This provides formal justification for the informal claims made by the algorithms' designers.

The formalism used in this paper is based on that in [9]. In [9], Lynch and Merritt develop a model for nested transaction systems including aborts, and use the model to show that an exclusive locking variation of Moss's algorithm [13] ensures correctness for non-orphans. In this paper we use the model to describe the orphan elimination algorithms, to state correctness properties, and to prove them correct.

Earlier work on verifying the Argus algorithm was done by Goree [4], who used a simple trace-based model for nested transaction systems [8]. While Goree was able to state and prove similar properties of the algorithm, the correctness properties were difficult to state and the proofs were extremely complex. In contrast, the correctness property described below is simple and intuitive, and our proofs are correspondingly simple. This provides strong evidence that the basic model from [9] is both simple and powerful enough for modelling and analyzing nested transaction systems.

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 relevant material from [9]. In particular, we define *generic systems*, which are nested transaction systems

in which orphans may occur, and for which the problem of managing orphans can be precisely and intuitively stated. A generic system models the components of a nested transaction system by I/O automata. Each user program is modelled as a transaction automaton, and each data item (together with whatever concurrency control and recovery mechanisms it uses) is modelled as a generic object automaton. The generic controller automaton links transactions and objects together, functioning as a minimally specified message-passing and scheduling mechanism. The material in these two sections is largely abstracted from [9]; except for Section 3.5; the reader who is familiar with [9] 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 insures orphan elimination by using a modified controller. The *filtered controller* maintains global information about the history of the system, and uses this to prevent accesses from committing if they could cause orphans to learn that they were orphans. The *Argus controller* maintains local information instead of global information, and models closely the behavior of the orphan elimination algorithm used in the Argus system to handle orphans that result from aborts [7]. The *strictly filtered controller* uses global information and is even more restrictive than the filtered controller. The *clock controller* models the algorithm from [11].

We prove that a *filtered system* (a system with the filtered controller) simulates the corresponding generic system so that all transactions, including orphans, "believe" they are part of the generic system, in an execution in which they are not orphans. From this it follows that the filtered controller ensures serial correctness if it is used with any concurrency control mechanism that ensures correctness for non-orphans in the generic system. We also prove that an *Argus system* simulates a filtered system, and so inherits the same correctness property. Similarly, we prove that a *clock system* simulates a *strictly filtered system*, which in turn simulates filtered system, thus showing that a clock system has the same correctness property as the filtered system.

After presenting the descriptions and proofs of the algorithms, we conclude in Section 9 with a summary of our results and some suggestions for further work.

2. Basic Model

We use the I/O automaton model [9, 10], 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 $\mathcal{A}$ has components *states*($\mathcal{A}$), *start*($\mathcal{A}$), *out*($\mathcal{A}$), *in*($\mathcal{A}$), and *steps*($\mathcal{A}$). Here, *states*($\mathcal{A}$) is a set of states, of which a subset, *start*($\mathcal{A}$), is designated as the set of start states. The next two components are disjoint sets: *out*($\mathcal{A}$) is the set of *output operations*, and *in*($\mathcal{A}$) is the set of *input operations*. The union of these two sets is the set of *operations* of the automaton. Finally, *steps*($\mathcal{A}$) is the transition relation of $\mathcal{A}$, 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 $\mathcal{A}$. If (s',π,s) is a step of $\mathcal{A}$, we say that π is *enabled* in s'.

The output operations are intended to 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 condition, which says that an I/O automaton must be prepared to receive any input operation at any time.

Input Condition: For each input operation π and each state s', there exist a state s and a step (s',π,s).

An *execution* of $\mathcal{A}$ is a finite alternating sequence $s_0,\pi_1,s_1,\pi_2,\ldots$ of states and operations of $\mathcal{A}$, ending with a state. Furthermore, s_0 is in start($\mathcal{A}$), and each triple (s',π,s) that occurs as a consecutive subsequence is a step of $\mathcal{A}$. 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 $\mathcal{A}$ is said to *preserve* S provided that the following holds. If $\alpha == \alpha'\pi$ is any schedule of $\mathcal{A}$, 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.

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* $\mathcal{S}$, if the sets of output operations for the automata are disjoint. (Thus, every output operation in $\mathcal{S}$ will be triggered by exactly one component.) The system $\mathcal{S}$ 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. Thus, each operation of the composed automaton is an operation of a subset of the component automata. 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. Again, the operation π is an output operation of the composition if it is the output operation of a component — otherwise, π is an input operation of the composition.

An *execution* of a system is defined to be an execution of the automaton composed of the individual automata of the system. 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 $\mathcal{A}$.

The following lemma from [9] expresses formally the idea that an operation is under the control of the component of which it is an output.

> **Lemma 1:** Let α' be a schedule of a system S, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathcal{A}$. If $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$, then α is a schedule of S.

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 [9]. 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 ($\mathcal{T}$,parent,$\mathcal{O}$,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. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and

descendant.) The leaves of this tree are called *accesses*. The set $\mathcal{O}$ denotes the set of objects; formally, $\mathcal{O}$ 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 tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be an infinite structure.

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. It is natural to reason about T_0 in much the same way as about all of the other 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 $\mathcal{O}$ 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 $\mathcal{O}$, and a generic controller. These automata are described below. (If X is a generic object associated with an element $\mathcal{X}$ of the partition $\mathcal{O}$, and T is an access in $\mathcal{X}$, we write $T \in accesses(\mathcal{X})$ and say that "T is an access to X".)

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

3.1. Transactions

Transactions are modelled as I/O automata. In modelling transactions, we consider it very important not to constrain them unnecessarily; thus, we do not want to require that they be expressible as programs in any particular high-level programming language. Modelling the transactions as I/O automata allows us to state exactly the properties that are needed, without introducing unnecessary restrictions or complicated semantics.

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' $\in$ children(T) and v $\in$ V

ABORT(T'), for T' $\in$ children(T)

Output operations:

REQUEST_CREATE(T'), for T' $\in$ children(T)
REQUEST_COMMIT(T,v), for v $\in$ 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.[5] 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, without returning any other information. 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, and includes a value recording the results of that work.

It is convenient to use two separate operations, REQUEST_CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST_CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. In actual distributed systems such as Argus [6], this separation does occur, and the distinction will be important in our results and proofs. Similar remarks hold for the REQUEST_COMMIT and COMMIT operations.

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 syntactic constraints. Thus, transaction automata are required to preserve well-formedness, as defined below.

We recursively define *well-formedness* for sequences of operations of transaction T. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following hold:

- If π is CREATE(T), then
  (i) there is no CREATE(T) in α'.

- If π is COMMIT(T',v) or ABORT(T') for a child T' of T, then
  (i) REQUEST_CREATE(T') appears in α' and
  (ii) there is no return operation for T' in α'.

- If π is REQUEST_CREATE(T') for a child T' of T, then
  (i) there is no REQUEST_CREATE(T') in α'

[5] Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include it in the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

(ii) there is no REQUEST_COMMIT(T) in α' and
(iii) CREATE(T) appears in α'.

- If π is a REQUEST_COMMIT for T, then
(i) there is no REQUEST_COMMIT for T in α' and
(ii) CREATE(T) appears in α'.

These restrictions are very basic; they simply say that a transaction does not get created more than once, does not receive repeated notification of the fates of its children, does not receive conflicting information about the fates of its children, and does not receive information about the fate of any child whose creation it has not requested; also, a transaction does not perform any output operations before it has been created or after it has requested to commit, and does not request the creation of the same child more than once. Except for these minimal conditions, there are no restrictions on allowable transaction behavior. For example, the model allows a transaction to request to commit without discovering the fate of all subtransactions whose creation it has requested. Also, a transaction can request creation of new subtransactions at any time, without regard to its state of knowledge about subtransactions whose creation it has previously requested. Particular programming languages may choose to impose additional restrictions on transaction behavior. (An example is Argus, which suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

3.2. Generic Objects

Generic objects are similar to the abstract objects of Argus and other "object-oriented" systems. A generic object provides a set of "operations" (not to be confused with the operations of an I/O automaton) through which transactions can observe and change the object's state. For uniformity and ease of exposition, we model each possible instance of an "operation" as a subtransaction, here called an *access transaction*. Accesses can be invoked by concurrent transactions, and transactions can abort; thus, generic objects must provide synchronization and recovery sufficient to ensure serializability of the transactions using them. For example, the particular objects studied in [9], which use an exclusive locking variation of Moss's algorithm [13] for synchronization combined with version stacks for recovery, have been shown to be correct for non-orphan transactions [9]. In section 3.5, we will discuss in more detail the various possible notions of correctness and what properties are ensured by the locking objects studied in [9].

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. Thus, generic objects are required to preserve well-formedness, defined recursively as follows: First, the empty schedule is well-formed. Second, if $\alpha = \alpha'\pi$ is a sequence of operations of X, then α is well-formed provided that α' is well-formed and the following hold:

- If π is CREATE(T), then
  (i) there is no CREATE(T) in α'.

- If π is a REQUEST_COMMIT for T, then
  (i) there is no REQUEST_COMMIT for T in α', and
  (ii) CREATE(T) occurs in α'.

- If π is INFORM_COMMIT_AT(X)OF(T), then
  (i) there is no INFORM_ABORT_AT(X)OF(T) in α', and
  (ii) if T is an access to X, then a REQUEST_COMMIT for T occurs in α'.

- If π is INFORM_ABORT_AT(X)OF(T), then
  (i) there is no INFORM_COMMIT_AT(X)OF(T) in α'.

These restrictions are again quite basic. They state that a given access is created at most once, and requests to commit at most once, and then only if it has been created. In addition, an object should not be given conflicting information about the fate of a transaction, i.e., it should not be told both that a transaction committed and that it aborted. Finally, an object X should be told that an access to X has committed only if the access actually requested to commit.

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 I/O automata whose operations and behavior satisfy simple syntactic restrictions. A generic controller, however, is a fully specified automaton, particular to each system type. (Recall that we have assumed that the system type is fixed; we describe the generic controller for the fixed system type.)

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) $\cup$ 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') $\cup$ {T}

- REQUEST_COMMIT(T,v)
  Postcondition:
  commit_requested(s) = commit_requested(s') $\cup$ {(T,v)}

- CREATE(T)
  Precondition:
  T $\in$ create_requested(s') — created(s')

Postcondition:
created(s) = created(s') $\cup$ {T}

- COMMIT(T,v)
  Precondition:
  (T,v) $\in$ commit_requested(s')
  T $\notin$ returned(s')
  children(T) $\cap$ create_requested(s') $\subseteq$ returned(s')
  Postcondition:
  committed(s) = committed(s') $\cup$ {T}

- ABORT(T)
  Precondition:
  T $\in$ create_requested(s') — returned(s')
  Postcondition:
  aborted(s) = aborted(s') $\cup$ {T}

- INFORM_COMMIT_AT(X)OF(T):
  Precondition:
  T $\in$ committed(s')

- INFORM_ABORT_AT(X)OF(T):
  Precondition:
  T $\in$ aborted(s')

The controller assumes that its input operations, REQUEST_CREATE and REQUEST_COMMIT, can occur at any time, and simply records them in the appropriate components of the state. Once the creation of a transaction has been requested, the controller can create it by producing a CREATE operation. The precondition of CREATE indicates that a given transaction will be created at most once; the postcondition of CREATE records the fact that the creation has occurred. Similarly, the postconditions for COMMIT and ABORT record that the operation has occurred. 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 (committed or aborted). In addition, the actual COMMIT operation must be delayed until all children requested by the committing transaction have returned. Recall from the well-formedness conditions described earlier that a transaction, once created, can request to commit at any time. However, once it has done so, it cannot request that any more children be created.

The precondition for the ABORT operation ensures that a transaction will be aborted only if a REQUEST_CREATE has occurred for it and it has not already returned. There are no other constraints on when a transaction can be aborted, however. For example, a transaction can be aborted

while some of its descendants are still running.

The generic controller presented here is a slight generalization of the "weak concurrent controller" in [9], in that here we permit transactions to be created after they abort. Since the results in [9] and their proofs are essentially unaffected by the generalization, we see no reason not to generalize in this way.

The following lemma states some simple invariants relating schedules of the generic controller to the states that result from applying them to the initial state.

Lemma 2: Let α be a schedule of the generic controller, and let s be a state that can result from applying α to the initial state s_0. Then the following conditions are true.

1. T is in create_requested(s) exactly if α contains a REQUEST_CREATE(T) operation.
2. T is in created(s) exactly if α contains a CREATE(T) operation.
3. (T,v) is in commit_requested(s) exactly if α contains a REQUEST_COMMIT(T,v) operation.
4. T is in committed(s) exactly if α contains a COMMIT operation for T.
5. T is in aborted(s) exactly if α contains an ABORT(T) operation.
6. aborted(s) $\cap$ committed(s) $= \emptyset$

Proof: Straightforward. □

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*.

Define the *generic operations* to be those operations that occur in the generic system: REQUEST_CREATEs, REQUEST_COMMITs, CREATEs, COMMITs, ABORTs, INFORM_COMMITs and INFORM_ABORTs. 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.)

A sequence of generic operations is called *well-formed* provided that its projection on each generic primitive (transaction and generic object) is well-formed.

Lemma 3: Every generic schedule is well-formed.

Proof: By induction on the length of schedules. The basis, when the length of α is 0, is trivial. Suppose that $\alpha\pi$ is a generic schedule, where π is a single operation, and assume that α is well-formed. It suffices to show that $\alpha\pi|P$ is well-formed for all generic primitives P. Let P be any fixed generic primitive. If π is not an operation of P, the result is immediate, so

assume that π is an operation of P. We consider cases.

1. π is an output of P
   Since generic primitives preserve well-formedness, the result is immediate.

2. π is CREATE(T)
   The generic controller preconditions and Lemma 2 imply that no CREATE(T) appears in α.

3. π is COMMIT(T,v)
   Then π is an input to the transaction parent(T). The generic controller preconditions and Lemma 2 imply that REQUEST_COMMIT(T,v) occurs in α. The well-formedness of α implies that CREATE(T) occurs in α. The generic controller preconditions and Lemma 2 imply that REQUEST_CREATE(T) occurs in α. Also, the generic controller preconditions and Lemma 2 ensure that α does not contain a return operation for T.

4. π is ABORT(T)
   Then π is an input to the transaction parent(T). The generic controller preconditions and Lemma 2 imply that REQUEST_CREATE(T) occurs in α, and also imply that no return operation for T occurs in α.

5. π is INFORM_COMMIT_AT(X)OF(T)
   By the generic controller preconditions and Lemma 2, there is no INFORM_ABORT_AT(X)OF(T) in α, and there is a COMMIT operation for T in α. Again by the generic controller preconditions and Lemma 2, a REQUEST_COMMIT for T occurs in α. If T is an access to X, this operation occurs at X.

6. π is INFORM_ABORT_AT(X)OF(T)
   By the generic controller preconditions and Lemma 2, there is no INFORM_COMMIT_AT(X)OF(T) in α.

□

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 [9] 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 [9] serve the same purpose as the "serial specifications" in [21].)

Serial schedules capture the notion that an aborted transaction has no effect, and that siblings execute sequentially. Thus, they serve as the basis against which correctness is defined for more complicated systems, such as generic systems.

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 $\beta|\mathrm{T} = \alpha|\mathrm{T}$. 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.[6] 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. (This is true of all published concurrency control algorithms for nested transactions of which the authors are aware.) 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 [9], 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, in particular, the processing of information about the fate (commit or abort) of each transaction. The lock manager implements an exclusive locking protocol based on that of Moss [13]. The combination can

[6] As discussed in [9], this definition of correctness allows different transactions in α to "see" different serial schedules. However, correctness applies to the root transaction as well, so the root must see the same results from the top-level transactions in a generic schedule that it could see in some serial schedule.

be encapsulated in a generic object, which handles both concurrency control and recovery.[7] 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*.[8] We call schedules of a locking system *locking schedules*. The following theorem expresses the correctness guarantee proved in [9] for locking systems.

Theorem 4: 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. We prove a result of the following sort for each orphan elimination algorithm: if α is a schedule of the system with orphan elimination and T is a transaction, then there exists a generic schedule β such that $\beta|\mathrm{T} = \alpha|\mathrm{T}$ and T is not an orphan in β. In other words, the orphan elimination algorithms prevent transactions from "knowing" that they are orphans — everything a transaction sees is consistent with what it could see in some execution in which it is not an orphan. We will make these ideas more precise later in the paper.

Theorem 4 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, however. 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 [21] 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 [17], and objects that use more general approaches to locking [5, 18, 21]. 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.

---

[7] The only difference is that the CREATE input operation has another name in [9].

[8] Such a system is called a "weak concurrent system" in [9].

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[9] 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 they both occur at the same primitive (and π is an output, since inputs can always occur), or if they involve different primitives but the preconditions for the controller require ϕ to occur before π can occur. This notion of one operation affecting another is "safe," in the sense that ϕ affects π if there is any way that the precondition for π could require ϕ to have occurred. If the operations involve different primitives, the preconditions for π do require ϕ to occur if ϕ directly-affects π. If the operations occur at the same primitive, however, it might be that ϕ happens to occur before π, yet that the particular primitive does not require ϕ to occur before π. In the absence of more information about the particular primitives used in a system, however, it is difficult to say more about the ways in which one operation can affect another. Thus, we make the "safe" choice of assuming an effect whenever one could occur. Fortunately, the orphan elimination algorithms described later in this paper are essentially independent of the particular primitives used in a system, and do not rely on more information about them.

The following lemmas follow directly from the definitions:

Lemma 5: Let α be a sequence of generic operations. If ϕ affects π in α, then there exists a $\psi \neq \pi$ such that ψ directly-affects π in α, and either $\psi = \phi$ or ϕ affects ψ in α.

---

[9] 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). To avoid introducing excessive and confusing notation, we will not be overly formal in distinguishing operations from operation instances. For example, we will write that an operation instance is CREATE(T), meaning formally that its second component is CREATE(T).

Lemma 6: Let α be a sequence of generic operations. If ϕ affects π in α, α' is a prefix of α, and ϕ and π both occur in α', then ϕ affects π in α'.

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 is immediate from the definitions:

Lemma 7: Let α be a sequence of generic operations, and let β be a closed subsequence of α. If β' is a prefix of β, then β' is closed in α.

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

Lemma 8: If α is a generic schedule, then any closed subsequence of α is also a generic schedule.

Proof: Fix α. We proceed by induction on the length of subsequences β of α, to show that, if β is closed in α, then β is a generic schedule. The basis, when the length of β is 0, is trivial. For the inductive step, assume that β is a closed subsequence of α of length at least 1. Then let $\beta = \beta'\pi$, where π is a single operation. Let α' be the prefix of α preceding π. By Lemma 7, β' is closed in α. Thus, the inductive hypothesis implies that β' is a generic schedule. We consider cases.

1. π is an output operation of a primitive P
   Since β is closed in α, it follows from the definition of affects(α) that β contains all operations of P that precede π in α. Thus, $\beta|P$ is a prefix of $\alpha|P$, and since $\alpha|P$ is a schedule of P, it must be that $\beta|P$ is also a schedule of P. Then β is a generic schedule, by Lemma 1.

2. π = CREATE(T)
   Then Lemma 2 and the controller preconditions imply that REQUEST_CREATE(T) occurs in α', and no CREATE(T) occurs in α'. Since β is closed in α, REQUEST_CREATE(T) also occurs in β'. Since no CREATE(T) occurs in β', Lemma 2 implies that π is enabled in the (unique) generic controller state resulting from β'. Thus, β is a schedule of the generic controller, so Lemma 1 implies that β is a generic schedule.

3. π = COMMIT(T,v)
   Then Lemma 2 and the controller preconditions imply that α' contains REQUEST_COMMIT(T,v) and contains no return operations for T. In addition, α' contains a return operation for each child T' of T for which REQUEST_CREATE(T') appears in α'. Since β is closed in α, β contains these operations (by the definition of directly-affects); thus, so does β'. Since no return operations for T occur in β', Lemma 2 implies that π is enabled after β', so β is a generic schedule.

4. π = ABORT(T)
   Then α' contains REQUEST_CREATE(T) and contains no return operations for T. Since β is closed in α, β' also contains REQUEST_CREATE(T). Since no return operations for T occur in β', π is enabled after β', so β is a generic schedule.

5. π = INFORM_COMMIT_AT(X)OF(T)
   Then α' contains a COMMIT operation for T. Since β is closed in α, so does β'. Thus,

π is enabled after β', so β is a generic schedule.

6. π = INFORM_ABORT_AT(X)OF(T)
Then α' contains an ABORT operation for T. Since β is closed in α, so does β'. Thus, π is enabled after β', so β is a generic schedule.

□

This lemma shows that the relation affects(α) captures all ways in which one operation can depend on another. 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.

The intuitive idea behind the orphan elimination algorithms is that they ensure that an operation of a transaction T is never affected by the abort of an ancestor. Once we have shown this, Lemma 8 allows us to show that every transaction gets a view it could get in an schedule in which it is not an orphan: we simply take the subsequence of the schedule containing all operations of T and all operations that affect them. The resulting sequence is a generic schedule, by Lemma 8, and does not contain an abort for an ancestor of T, by construction.

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 is the same as the generic controller, except that it permits an access to commit only if the commit is not affected by the abort of an ancestor.

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 (i.e., create_requested(s), created(s), commit_requested(s), committed(s), and aborted(s)). 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')
  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') $\cup$ {T}
  history(s) = history(s')COMMIT(T,v)

Thus, at the point where an access is about to commit to its parent, an explicit test is performed to verify that the new COMMIT operation is not affected (in our formal sense) by the abort of any ancestor of the access.

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 9: Every filtered schedule is a generic schedule.

Proof: First we note the following claim. Suppose that α is both a filtered schedule and a generic schedule. Suppose that s_F is the (uniquely defined) state of the filtered controller after α and s_G is the (uniquely defined) state of the generic controller after α. Then s_G is the same as s_F except for the omission of the history component. The truth of this claim is easily seen by induction on the length of α.

Now we show the result by induction on the length of filtered schedules. The basis, length 0, is trivial. Let $\alpha = \alpha'\pi$ be a filtered schedule of length at least 1, where π is a single operation. If π is an output of a transaction or generic object P, then $\alpha|P$ is a schedule of P, and so the inductive hypothesis and Lemma 1 imply that α is a generic schedule. So assume that π is an output of the filtered controller. Let s_F be the state of the filtered controller after α', and let s_G be the state of the generic controller after α'. By the inductive hypothesis and the claim above, s_G is the same as s_F except for the deletion of the history component. Since π is enabled in s_F for the filtered controller, it is also enabled in s_G for the generic controller.

Thus, α is a schedule of the generic controller, and hence, by Lemma 1, 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 10: 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.

Proof: First note that Lemmas 9 and 3 imply that α is well-formed; we will use this fact in the proof of the lemma. The proof is by induction on the length of α. If α is empty, the result clearly holds. Suppose $\alpha = \alpha'\pi$, and that the lemma holds for α'. Obviously, affects(α) $\subseteq$ affects(α') $\cup$ $\{(\phi,\pi) \mid \phi$ is an operation in $\alpha'\}$. Thus, by induction, it suffices to show that the lemma holds when $\rho = \pi$. If location(π) is an object, the result is obvious, so assume that location(π) = T, where T is a non-access transaction.

Suppose that the lemma does not hold when $\rho = \pi$, i.e., that ϕ = ABORT(T') affects π in α, for some ancestor T' of T. We derive a contradiction.

By Lemma 5, there exists a $\psi \neq \pi$ such that ψ directly-affects π in α, and either $\psi = \phi$ or ϕ affects ψ in α. Since $\psi \neq \pi$, ϕ and ψ must occur in α'. Thus, by Lemma 6, if $\phi \neq \psi$, ϕ affects ψ in α'.

Notice that location(ϕ) = parent(T'); since T' is an ancestor of T, parent(T') $\neq$ T. Thus, location(ϕ) $\neq$ T.

We consider cases.

1. π is an output operation of T.
   Then by the definition of directly-affects(α), location(ψ) = T. Since location(ϕ) $\neq$ T, $\phi \neq \psi$. Thus, ϕ affects ψ in α'. But this contradicts the inductive hypothesis.

2. π is CREATE(T).
   Then by the definition of directly-affects(α), ψ = REQUEST_CREATE(T). Thus, $\phi \neq \psi$, so ϕ affects ψ in α'. Since REQUEST_CREATE(T) is an operation of parent(T), the inductive hypothesis implies that T' is not an ancestor of parent(T). The only possibility is that T' = T, which implies that ABORT(T) precedes REQUEST_CREATE(T) in α. But this implies that α|T is not well-formed, a contradiction.

3. π is COMMIT(T'',v), where T'' is a child of T and T'' is an access.
   Then the precondition for COMMIT(T'',v) in the filtered controller is violated, a contradiction.

4. π is COMMIT(T'',v), where T'' is a child of T and T'' is a non-access transaction.
   Then T' is an ancestor of T'', and by the definition of directly-affects(α), ψ is either REQUEST_COMMIT(T'',v) or a return operation for a child U of T''. Thus, $\phi \neq \psi$, so ϕ affects ψ in α', and location(ψ) is either T'' or U. But this contradicts the inductive hypothesis.

5. π is ABORT(T''), where T'' is a child of T.
Then by the definition of directly-affects(α), ψ = REQUEST_CREATE(T''), and location(ψ) = T. Thus, $\phi \not= \psi$, so ϕ affects ψ in α'. Again, this contradicts the inductive hypothesis.

□

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 11: 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 8, 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 10, this is impossible. □

As discussed earlier, we can combine Theorem 11 with Theorem 4 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 12: Any filtered locking system is serially correct.

Proof: Let α be a filtered locking schedule and let T be a non-access transaction. Theorem 11 yields a locking schedule γ such that T is not an orphan in γ and $\gamma|$T = $\alpha|$T. Theorem 4 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. If S is any generic system, define *filter*(S) to be the system obtained from S by replacing the generic controller in S with the filtered controller. The following is an immediate consequence of Theorem 11.

Corollary 13: If S is a generic system whose schedules are serially correct for non-orphan non-access transactions, then *filter*(S) is serially correct.

At first it might seem somewhat surprising that it is enough to prevent the commits of orphaned accesses to ensure serial correctness for all orphans. The reason it is not necessary to filter operations of other transactions is because of the restricted communication patterns among the primitives in a system. 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. In effect, by preventing the commits of orphaned accesses, we isolate orphaned transactions from the objects, ensuring that an orphaned transaction never sees that it is an orphan.

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[10] 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. In Argus, for example, each operation may occur only at a single physical node of the network, and each node manages a single set of aborted transactions for the entire set of operations that occur at that node. In this case, done(s)(π) includes at least the transactions whose aborts affect π to date, as well as those transactions whose aborts affect any other operation that has occured at the same physical node as π.)

[10] Note the technicality that the domain of done(s) is the set of operations, not the set of operation instances.

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. As before, we define returned(s) = committed(s) $\cup$ aborted(s).

The transition relations for the operations of the Argus controller are defined as follows. As usual, s' indicates the state before the indicated operation, and s indicates the state after the operation. To avoid repetition, we indicate once here that *every* operation π contains the following additional postcondition: for every operation ϕ (including π), done(s')(ϕ) $\subseteq$ done(s)(ϕ).

- REQUEST_CREATE(T)
  Postcondition:
  create_requested(s) = create_requested(s') $\cup$ {T}
  for all π such that location(π) = parent(T),
  done(s')(π) $\subseteq$ done(s)(REQUEST_CREATE(T))

- REQUEST_COMMIT(T,v)
  Postcondition:
  commit_requested(s) = commit_requested(s') $\cup$ {(T,v)}
  for all π such that location(π) = location(REQUEST_COMMIT(T,v)),
  done(s')(π) $\subseteq$ done(s)(REQUEST_COMMIT(T,v))

- CREATE(T)
  Precondition:
  T $\in$ create_requested(s') — created(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}
  done(s')(REQUEST_CREATE(T)) $\subseteq$ done(s)(CREATE(T))

- COMMIT(T,v), T a non-access
  Precondition:
  (T,v) $\in$ commit_requested(s')
  T $\notin$ returned(s')
  children(T) $\cap$ create_requested(s') $\subseteq$ returned(s')
  Postcondition:
  committed(s) = committed(s') $\cup$ {T}
  for all π such that π = REQUEST_COMMIT(T,v) or π is the return of a child of T,
  done(s')(π) $\subseteq$ done(s)(COMMIT(T,v))

- COMMIT(T,v), T an access
  Precondition:
  (T,v) $\in$ commit_requested(s')
  T $\notin$ returned(s')
  there is no ancestor of T in done(s')(REQUEST_COMMIT(T,v))
  Postcondition:
  committed(s) = committed(s') $\cup$ {T}
  done(s')(REQUEST_COMMIT(T,v)) $\subseteq$ done(s)(COMMIT(T,v))

- ABORT(T)
  Precondition:
  T $\in$ create_requested(s') — returned(s')

Postcondition:
aborted(s) = aborted(s') ∪ {T}
done(s')(REQUEST_CREATE(T)) ∪ {T} ⊆ done(s)(ABORT(T))

- INFORM_COMMIT_AT(X)OF(T):
  Precondition:
  T ∈ committed(s')
  Postcondition:
  for all v such that (T,v) ∈ commit_requested(s'),
  done(s')(COMMIT(T,v)) ⊆ done(s)(INFORM_COMMIT_AT(X)OF(T))

- INFORM_ABORT_AT(X)OF(T):
  Precondition:
  T ∈ aborted(s')
  Postcondition:
  done(s')(ABORT(T)) ⊆ done(s)(INFORM_ABORT_AT(X)OF(T))

There are two significant differences between the Argus controller and the generic controller. First, the postconditions for each operation π in the Argus controller require done(s)(π) to include done(s')(ϕ) for each ϕ that directly-affects π.[11] In addition, the postcondition for ABORT(T) requires T to be in done(s)(ABORT(T)). As Lemma 16 below shows, these constraints are enough to ensure that done(s)(π) contains T whenever ABORT(T) affects an instance of π.

Second, the precondition for the commit of an access permits the commit to occur only if the access does not "know about" the abort of an ancestor, i.e., no ancestor is in done(s')(REQUEST_COMMIT(T,v)). As Lemma 17 below shows, this is enough to ensure that every Argus schedule is a filtered schedule.

The mapping done(s) models the distributed information maintained by the Argus algorithm to keep track of actions that abort. However, rather than modelling nodes directly and keeping the information on a per-node basis as is done in the actual algorithm, we maintain the information for each operation, propagating it whenever one operation directly-affects another.

The controller postconditions are stated so as to insure that at least the minimum amount of necessary information is propagated at each step. An implementation is permitted to propagate more than the minimum; for instance, an implementation might keep track of the done mapping at a coarser granularity. (By maintaining the done mapping on a per-node basis, the implementation of the Argus algorithm in the current Argus prototype follows this strategy.) In describing the algorithm, we have tried to focus on the behavior necessary for correctness, and to avoid constraining an implementation any

[11] The postcondition for the INFORM_COMMIT operation may be a little confusing. Since the value returned in the COMMIT operation is not part of the INFORM_COMMIT, the postcondition requires done(s')(COMMIT(T,v)) to be included for all (T,v) in commit_requested(s'). For a given T, however, only one COMMIT(T,v) will occur, so there will only be one such pair in commit_requested(s').

more than necessary.

Notice also that the controller does not put any upper limit on what goes into the done mapping. For example, as described above it is permissible for done(s)(π) to contain a transaction that has not aborted. It would be easy to add a requirement that done(s)(π) $\subseteq$ aborted(s), but this is not necessary to prove that the algorithm eliminates all orphans. To prove other properties, such as that the algorithm only detects and eliminates real orphans, we would need to add additional requirements such as the one just mentioned. We will not attempt to state or prove such properties in this paper; the property just described is a special case of more general liveness properties, which are the subject of current research.

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 14: Every Argus schedule is a generic schedule.

Proof: The proof is similar to that of Lemma 9. □

Lemma 15: Let $\alpha\pi$ be an Argus schedule, with π a single operation instance, and let s' and s be states of the Argus controller after α and $\alpha\pi$, respectively, such that (s',π,s) is a step of the controller. Let ϕ be any operation.

1. If an instance of operation ϕ directly affects π, an instance of ψ, in $\alpha\pi$, then done(s')(ϕ) $\subseteq$ done(s)(ψ).

2. done(s')(ϕ) $\subseteq$ done(s)(ϕ).

3. If π is an instance of ABORT(T), then T $\in$ done(s)(ABORT(T)).

Proof: Immediate. □

Lemma 16: 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).

Proof: The proof proceeds by induction on the length of α. If α is of length 0, there is nothing to prove. So suppose $\alpha = \alpha'\pi$, where π is a single operation instance, and that the lemma holds for α'. Let s' be a state of the Argus controller after α' such that (s',π,s) is a step of the Argus controller. Fix ϕ, an operation with an instance in α that is affected by an instance of ABORT(T).

Note first that a simple induction using parts 2 and 3 of Lemma 15 suffices to show that T $\in$ done(s')(ABORT(T)).

If an instance of ϕ is affected by an instance of ABORT(T) in α', the result follows by the inductive hypothesis and part 2 of Lemma 15.

Otherwise, π is an instance of ϕ, and by Lemma 5, an instance of ABORT(T) either directly affects π in α, or affects an operation instance ψ in α (and hence in α'), which in turn directly affects π in α. In the first case, part 1 of Lemma 15 implies done(s')(ABORT(T)) $\subseteq$ done(s)(ϕ), and by the observation above T $\in$ done(s')(ABORT(T)), so T $\in$ done(s)(ϕ). In the latter case, the inductive hypothesis implies that T $\in$ done(s')(ψ), and part 1 of Lemma 15

implies done(s')(ψ) $\subseteq$ done(s)(ϕ). □

6.3. Simulation of Generic Systems by Argus Systems

Lemma 16 shows that the Argus controller propagates enough information about aborts so that every operation π "knows about" (stores in done(s)(π)) every abort that affects it. 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 17: Every Argus schedule is a filtered schedule.

Proof: The proof is by induction on the lengths of Argus schedules. The basis, length 0, is trivial. For the inductive step, let α be an Argus schedule of the form $\alpha'\pi$, where π is a single operation. Let s_A be a state of the Argus controller after α', and let s_F be the state of the filtered controller after α'. (Notice that the state of the filtered controller is uniquely defined by α', while the state of the Argus controller is not.) Examining the usual cases, the only one that is not immediate is where π = COMMIT(T,v) and T is an access. So assume that this is the case.

We must show that π is enabled in s_F. This amounts to showing that if T' is an ancestor of T, then ABORT(T') does not affect π in α. Suppose the contrary: that ϕ = ABORT(T'), for some ancestor T' of T, and that ϕ affects π in α. Since T has no children, ϕ affects ψ = REQUEST_COMMIT(T,v) in α'. By Lemma 16, T' $\in$ done(s_A)(ψ). But this violates the precondition for π in the Argus controller. □

The following theorem shows that Argus systems, like filtered systems, ensure that every non-access transaction gets a view it could get in an execution in which it is not an orphan.

Theorem 18: . Let α be an Argus 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: Immediate by Lemma 17 and Theorem 11. □

As for the filtered controller, if we define an *Argus locking system* to be an Argus system with locking objects, and *Argus locking schedules* to be schedules of Argus locking systems, then we obtain the following corollary.

Corollary 19: Any Argus locking system is serially correct.

Proof: Let α be an Argus locking schedule and let T be a non-access transaction. Theorem 18 yields a locking schedule γ such that T is not an orphan in γ and $\gamma|T = \alpha|T$. Theorem 4 then yields the required serial schedule β. □

Similarly, we can use the Argus controller with any collection of objects that ensures serial correctness for non-orphans, and obtain serial correctness for all non-access transactions.

7. Strictly Filtered Systems

The orphan elimination algorithm described in [11] actually ensures a stronger property than does the Argus algorithm. It ensures that no orphan runs at an object if an ancestor has aborted, whereas the Argus algorithm merely ensures that no orphan observes that an ancestor has aborted. (If the system is serially correct for non-orphans, we believe that there is no observable difference between these two properties.) In this section we define a "strictly filtered controller", which allows an access to commit only if no ancester has aborted. (Compare this to the filtered controller, which allows an access to commit if an ancestor has aborted as long as the access is not affected by the abort.) We then define strictly filtered systems, which are composed of transactions, generic objects, and the strictly filtered controller, and show that strictly filtered systems simulate filtered systems. In the next section we will describe formally the algorithm from [11] and show that it simulates strictly filtered systems.

7.1. Strictly Filtered Controller

The strictly filtered controller is similar to the generic controller: it has the same operations, and the same states. The transition relations associated with the operations are also identical to those for the generic controller, except for the COMMIT operation for accesses, which is defined as follows:

- COMMIT(T,v), T an access
  Precondition:

  $(T,v) \in commit\_requested(s')$
  $T \notin returned(s')$
  $ancestors(T) \cap aborted(s') = \emptyset$

  Postcondition:

  $committed(s) = committed(s') \cup \{T\}$

The COMMIT operation for an access has an additional precondition, which permits the transaction to commit only if none of its ancestors has already aborted.

7.2. Strictly Filtered Systems

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

Lemma 20: Every strictly filtered schedule is a generic schedule.

Proof: Immediate. □

7.3. Simulation of Generic Systems by Strictly Filtered Systems

Lemma 21: Every strictly filtered schedule is a filtered schedule.

Proof: By induction on the length of strictly filtered schedules. The basis, when the length of the schedule is 0, is easy. For the inductive step, let $\alpha = \alpha'\pi$ be a strictly filtered schedule, with π a single operation. Let s_S be the state of the strictly filtered controller after α', and let

s_F be the state of the filtered controller after α'. The only difference between s_S and s_F is that s_F includes α' as its history component.

The only interesting case is where π = COMMIT(T,v), for T an access, so assume this is so. Since π is enabled in s_S, ancestors(T) $\cap$ aborted(s_S) = $\emptyset$, so that there is no ABORT(T') in α', for T' an ancestor of T. Then no such ABORT(T') can affect π in α, so π is enabled in s_F. □

Like filtered systems and Argus systems, strictly filtered systems prevent orphans from discovering that they are orphans:

Theorem 22: Let α be a strictly 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: Immediate by Lemma 21 and Theorem 11. □

As before, we can define a *strictly filtered locking system* to be a strictly filtered system with locking objects, and *strictly filtered locking schedules* to be schedules of strictly filtered locking systems. We then obtain the following corollary.

Corollary 23: Any strictly filtered locking system is serially correct.

Proof: Let α be a strictly filtered locking schedule and let T be a non-access transaction. Theorem 22 yields a locking schedule γ such that T is not an orphan in γ and $\gamma|T = \alpha|T$. Theorem 4 then yields the required serial schedule β. □

As before, we can use the strictly filtered controller with any collection of objects that ensures serial correctness for non-orphans, and obtain serial correctness for all non-access transactions.

8. Clock Systems

In this section we describe formally the orphan elimination algorithm from [11]. (The algorithm described here actually generalizes the one described in [11]. The algorithm in [11] delays commits of transactions, as well as aborts, while the algorithm described here delays only aborts. A similar generalization is described in [12].) We do this by defining a "clock controller," which uses a global clock to ensure that transactions do not abort until all their descendant accesses have stopped running. We then define clock systems, which are composed of transactions, generic objects, and the clock controller. Finally, we show that clock systems simulate strictly filtered systems, and thus generic systems as well.

8.1. The 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.

If quiesce and release times are fixed in advance, some transactions may be forced to abort unnecessarily

as their quiesce times expire. It is possible to obtain extra flexibility by providing operations in the clock controller for adjusting quiesce and release times.

The clock controller has ten 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)
TICK
ADJUST_QUIESCE(T)
ADJUST_RELEASE(T)

These are the same as for the generic controller, with the addition of three new output operations: TICK, ADJUST_QUIESCE and ADJUST_RELEASE. These output operations are to be thought of as "internal" to the clock controller, in that they will not be used as inputs to any other components of the clock system. The TICK operation advances the clock, while the two ADJUST operations adjust quiesce and release times. By adjusting the quiesce time for a transaction to be later than its current value, we can extend the time during which a transaction is allowed to run. Similarly, by adjusting the release time for a transaction to be earlier than its current value, we can allow a transaction to abort without waiting as long as would otherwise be necessary.

The state of the clock controller consists of eight components. The first five are as in the generic controller, and are initialized in the same way. The other three are clock(s), quiesce(s), and release(s). As before, we denote the initial state of the controller by s_0. Clock(s) is a real number, initialized arbitrarily. Quiesce(s) is a total mapping from access transactions to real numbers, and release(s) is a total mapping from all transactions to real numbers. The initial values of quiesce and release are arbitrary, subject to the following condition: for all transactions T and T', where T is an access and T' is an ancestor of T, $\text{quiesce}(s_0)(T) \leq \text{release}(s_0)(T')$. We define returned(s) as usual.

The transition relations associated with the clock controller operations are as for the generic controller, except for COMMIT(T,v) for access transactions T, ABORT(T), TICK, ADJUST_QUIESCE(T) and ADJUST_RELEASE(T). These are defined below.

- COMMIT(T,v), where T is an access
  Precondition:

(T,v) $\in$ commit_requested(s')
T $\notin$ returned(s')
clock(s') $<$ quiesce(T)(s')
Postcondition:
committed(s) = committed(s') $\cup$ {T}

- ABORT(T)
  Precondition:
  T $\in$ create_requested(s') — returned(s')
  release(s')(T) $\leq$ clock(s')
  Postcondition:
  aborted(s) = aborted(s') $\cup$ {T}

- TICK
  Postcondition:
  clock(s') $<$ clock(s)

- ADJUST_RELEASE(T)
  Precondition:
  if T $\in$ aborted(s'), then r $\leq$ clock(s')
  quiesce(s')(T') $\leq$ r for all T' $\in$ descendants(T)
  Postcondition:
  release(s)(T) = r

- ADJUST_QUIESCE(T)
  Precondition:
  q $\leq$ release(s')(T') for all T' $\in$ ancestors(T)
  Postcondition:
  quiesce(s)(T) = q

8.2. Clock Systems

A *clock system* is the composition of transactions, generic objects, and the clock controller. Operations of a clock system are called *clock operations*. Schedules of a clock system are called *clock schedules*. If α is any sequence of clock operations, define *generic*(α) to be the subsequence of α containing exactly the generic operations in α.

Lemma 24: If α is a clock schedule, then generic(α) is a generic schedule.

Proof: Straightforward by induction. □

Lemma 25: Let α be a clock schedule, and let s be a state of the clock controller after α. (a) If T $\in$ aborted(s), then release(s)(T) $\leq$ clock(s). (b) For all access transactions T and all ancestors T' of T, quiesce(s)(T) $\leq$ release(s)(T').

Proof: Straightforward by induction. □

8.3. Clock Systems Simulate Generic Systems

The following lemma shows that clock systems simulate strictly filtered systems.

Lemma 26: If α is a clock schedule, then generic(α) is a strictly filtered schedule.

Proof: The proof is by induction on the length of α. The basis, when the length of α is 0, is easy. For the inductive step, let $\alpha = \alpha'\pi$, where π is a single operation. Let s_C be a state of

the clock controller after α', and s_S the state of the strictly filtered controller after generic(α'). The only interesting case is where π = COMMIT(T,v) for T an access, so suppose this is so.

Since π is enabled in s_C, we know that clock(s_C) < quiesce(s_C)(T). Let T' be an ancestor of T. If T' $\in$ aborted(s_S), then T' $\in$ aborted(s_C). Then Lemma 25 implies that release(s_C)(T') $\leq$ clock(s_C). Lemma 25 also implies that quiesce(s_C)(T) $\leq$ release(s_C)(T'). Thus, quiesce(s_C)(T) $\leq$ clock(s_C), a contradiction. It follows that no ancestor of T is in aborted(s_S), so that π is enabled in s_S. □

Clock systems also prevent orphans from discovering that they are orphans:

Theorem 27: Let α be a clock schedule and T 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: Lemma 26 and Theorem 22 imply the existence of a generic schedule β such that T is not an orphan in β, and $\beta|T$ = generic(α)|T. But generic(α)|T = $\alpha|T$, so the result follows. □

We define a *clock locking system* to be a clock system with locking objects, and *clock locking schedules* to be schedules of clock locking systems. The following corollary is immediate.

Corollary 28: Any clock locking system is serially correct.

Proof: By Theorems 27 and 4. □

The algorithm described here uses a single physical clock to detect and eliminate orphans. The algorithm can be adapted to work with distributed, loosely synchronized physical clocks, or with logical clocks (e.g., see [12]). The adapted algorithms can be described and analyzed in a manner similar to that used for the Argus algorithm.

9. Conclusions

We have defined correctness properties for orphan elimination algorithms, and have presented precise descriptions and proofs for two algorithms from [7] and [11]. 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, 11] 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.

10. Acknowledgements

We thank Alan Fekete, Ken Goldman and Sharon Perl for their comments on earlier versions of this work.

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.
Concurrency control for resilient nested transactions.
Advances in Computing Research 3:335-373, 1986.

[9] 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.

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

[11] 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.

[12] McKendry, M., and Herlihy, M.
Timestamp-based orphan elimination.
Technical Report CMU-CS-87-108, Carnegie-Mellon University, 1987.

[13] 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.

[14] Nelson, B. J.
Remote procedure call.
PhD thesis, Carnegie-Mellon University Department of Computer Science, May, 1981.
Available as CMU-CS-81-119.

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

[16] 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.

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

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

[19] 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.

[20] Walker, E. F.
Orphan Detection in the Argus System.
Master's thesis, MIT, May, 1984.
Available as MIT-LCS-TR-326.

[21] 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.