On the Correctness of Orphan Management Algorithms

MAURICE HERLIHY

Digital Equipment Corporation, Cambridge, Massachusetts

NANCY LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

MICHAEL MERRITT

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

WILLIAM WEIHL

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. In a distributed system, node failures, network delays, and other unpredictable occurrences can result in *orphan* computations—subcomputations that continue to run but whose results are no longer needed. Several algorithms have been proposed to prevent such computations from seeing inconsistent states of the shared data. In this paper, two such orphan management algorithms are analyzed. The first is an algorithm implemented in the Argus distributed-computing system at MIT, and the second is an algorithm proposed at Carnegie-Mellon. The algorithms are described formally, and complete proofs of their correctness are given.

The proofs show that the fundamental concepts underlying the two algorithms are very similar in that each can be regarded as an implementation of the same high-level algorithm. By exploiting

---

M. Herlihy was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 (Amendment 20), under Contracts F33615-84-K-1520 and F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. N. Lynch was supported by the National Science Foundation under Grants DCR 83-02391 and CCR 86-11442, 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 the Office of Army Research under Contract DAAG29-84-K-0058. W. Weihl was supported by an IBM Faculty Development Award, the National Science Foundation under grants DCR 85-100014 and CCR 87-16884, and the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

Authors' addresses: M. Herlihy, Digital Equipment Corporation, Cambridge Research Laboratory, 1 Kendall Square, Cambridge, MA 02139; N. Lynch, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139; M. Merritt, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974; W. Weihl, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

properties of information flow within transaction management systems, the algorithms ensure that orphans only see states of the shared data that they could also see if they were not orphans. When the algorithms are used in combination with any correct concurrency control algorithm, they guarantee that all computations, orphan as well as nonorphan, see consistent states of the shared data.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; *specification techniques*; H.2.3 [**Database Management**]: Languages—*database (persistent) programming languages*; H.2.4 [**Database Management**]: Systems—*concurrency*; *distributed systems*; *transaction processing*

General Terms: Algorithms, Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Argus, avalon, atomic actions, camelot, input–output automata, recovery, serializability

1. *Introduction*

Nested transaction systems have been explored in a number of recent research projects (e.g., see [1], [8], [9], [21], and [23]) as a means for organizing computations in distributed systems. Nested transactions, like ordinary transactions, provide a simple construct for masking the effects of concurrency and failures. Nested transactions extend the usual notion of transactions [3] to permit concurrency within a single transaction. They also provide a greater degree of fault-tolerance by isolating a transaction from the failures of its descendants.

In distributed systems, various factors, including node crashes and network delays, can result in *orphan* computations—subcomputations that continue to run even though their results are no longer needed. For example, in the Argus system [9], a node making a remote request may give up because a network partition or some other problem prevents it from communicating 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 will not have any permanent effects on the observed state of the shared data.

As discussed in [10] and [17], 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 states of the shared data. For example, a transaction might read data at two nodes, with some invariant relating the values of the different data objects. 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. Although 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 interacts with the external world; this can 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 [19], which describes algorithms for detecting and eliminating orphans that arise because of node crashes. Nelson's work did not assume an underlying transaction mechanism, so it was difficult to assign simple semantics to abandoned computations. Recent work [10, 17, 24] 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 [10], [17], and [24] is to detect and eliminate orphans before they can see inconsistent information.

1.1. NEW RESULTS. In this paper, we give formal descriptions and correctness proofs for the two orphan management algorithms in [10] and [17]. The algorithm in [10] is currently in use in the Argus system.[1] Our proofs are completely rigorous, yet straightforward. In addition, both the presentations and the proofs follow the intuitions that the designers have used in describing the algorithms. Although the two algorithms appear to be quite different, our proofs show that the fundamental concepts underlying them are very similar; in fact, each can be regarded as an implementation of the same high-level algorithm.

Our results relate the behavior of a system, S′, containing an orphan management algorithm to that of a corresponding system, S, having no orphan management; namely, S′ must "simulate" S in the sense that each transaction in S′ must see a view of the system that it could see in an execution of S in which it is not an orphan. (A transaction's "view" of the system is its sequence of interactions with the system, including the results of operations and subtransactions invoked by the transaction.) When system S includes a concurrency control algorithm that ensures that nonorphans see consistent views, our results imply that in S′, *all* transactions, orphan as well as nonorphan, see consistent views. These results provide formal justification for informal claims sometimes made by the algorithms' designers that the algorithms work in combination with any concurrency control algorithm.

The formal model used in this paper is based on that in [4], [12], and [13]. In [12] and [13], 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 [18] ensures correctness for nonorphans. The paper [4] contains improvements to the basic model in [12] and [13], plus proofs that Moss's read–write algorithm and a more general commutativity-based locking algorithm also ensure correctness for nonorphans. In this paper, we use the same model to describe the two orphan management algorithms mentioned above, to state correctness properties, and to prove the algorithms correct.

1.2. RELATED WORK. Earlier work on verifying the Argus orphan management algorithm appears in [6]. This work is based on an earlier model for nested transaction systems that is described in [11]. The results in [6] are less general than those presented here, since they apply only to the specific

[1] Our analysis covers only orphans resulting from aborts of transactions that leave running descendants; there is another component of the Argus algorithm that handles orphans that result from node crashes in which the contents of volatile memory are destroyed.

concurrency control algorithm (nested locking) used by Argus. Moreover, the presentation there is much more complex than the one in this paper. Much of the complexity in [6] arises because the treatments of concurrency control and orphan management are intermingled, whereas here we are able to separate the two. The model in [4], [12], and [13] provides a convenient set of concepts for describing this separation.

Other work using the model of [4], [12], and [13] includes [2], [5], and [20]; these papers prove correctness of algorithms for replica management, timestamp-based concurrency control, and distributed transaction commit, respectively. The fact that it is possible to use the model to explain such a variety of transaction-processing algorithms is strong evidence that it is a useful tool for modeling and analyzing nested transaction systems.

In an earlier version of this paper [7], we presented similar results proving the correctness of the two algorithms analyzed here. The results in that paper, however, were restricted to a particular class of systems appropriate for modeling locking algorithms, and did not apply directly to systems using timestamps and other mechanisms. In this paper, we generalize the results of the earlier paper to apply to a wide range of systems. The general results described here are somewhat abstract; to make them more concrete, we give two examples illustrating how they apply to specific kinds of systems.

1.3. ORGANIZATION OF THIS PAPER. The remainder of the paper is organized as follows: Section 2 contains some preliminary mathematical definitions and a brief description of *I/O automata*, which serve as the formal foundation for our work. This section may be skipped on first or cursory reading, and is included in order to make the technical presentation of this paper entirely self-contained. Section 3 contains a definition of *basic systems*, a general class of transaction-processing systems to which our results apply. These are nested transaction systems in which orphans may occur, and for which the problem of managing orphans can be precisely and intuitively stated. A basic system models the components of a nested transaction system as I/O automata. Each user program is modeled as a transaction automaton, and the rest of the system (which may include a division into objects, and may include concurrency control and recovery algorithms) is modeled as a single basic database automaton. A basic system is said to mange orphans correctly if it ensures a property called *serial correctness* for all transactions, orphans, and nonorphans.

Section 4 contains some definitions and results about the dependencies among different events in a basic system; these concepts underlie the results in the rest of the paper.

Sections 5 through 8 contain the principal contributions of this paper, in which we prove the correctness of the two orphan management algorithms in [10] and [17]. Our proofs have an interesting structure. We first define a simple abstract algorithm that uses global information about the history of the system, and show that it ensures that orphans see consistent views. We then formalize the Argus algorithm and the clocked algorithm from [17] in a way that requires the use of local information only, and show that each simulates the more abstract algorithm. The simulation proofs are quite simple, and do not require reproving the properties already proved for the abstract algorithm. The correctness of the Argus and clocked algorithms then follows directly from the correctness of the abstract algorithm.

Each orphan management algorithm is described as a system obtained by transforming an arbitrary basic system without orphan management. Each of these systems contains the same transactions as the given basic system, but each manages orphans using a different basic database. The abstract algorithm is modeled by the *filtered database*, which maintains information about the global history of the system, and uses tests based on this history information to prevent orphans from learning that they are orphans. The *Argus database* models the behavior of the Argus orphan management algorithm [10]; it manages orphans using tests based on local information about direct dependencies among system events. The *strictly filtered database* models another abstract algorithm, introduced to simplify the proof of the correctness of the algorithm in [17]; it also uses tests based on global history information, and is even more restrictive than the filtered database. Finally, the *clock database* models the orphan management algorithm from [17]; it manages orphans using information about logical clocks. Each of these four databases is described as the result of a transformation of the basic database.

We prove that the *filtered system* (the system consisting of the transactions and the filtered database) simulates the basic system in the sense that all transactions, including orphans, see a "view" that they could see in the basic system in an execution in which they are not orphans. It follows that if the basic system ensures serial correctness for nonorphan transactions, then the filtered system ensures serial correctness for all transactions. We also prove that the *Argus system* implements the filtered system, and so inherits the same correctness property. Similarly, we prove that the *clock system* implements the *strictly filtered system*, which in turn implements the filtered system, thus showing that the clock system has the same correctness property as the filtered system.

Section 9 makes some of the preceding general concepts more concrete by describing two particular types of basic systems, taken from other work using this model. The first kind of basic system, a *generic system*, is appropriate for describing locking algorithms, while the second kind of basic system, a *pseudotime system*, is appropriate for describing timestamp-based algorithms. Both kinds of systems specialize the notion of a basic system by splitting the basic database automaton into two kinds of components: an object automaton for each object in the system and a controller automaton that links the transactions and objects together. The concurrency control and recovery performed by the system is encapsulated within the object automata. The two kinds of systems differ in that they have slightly different interfaces between the objects and the controller. Particular information flow dependencies are described for both of these kinds of basic systems.

Section 10 contains a summary of our results and some suggestions for further work.

2. *Formal Preliminaries*

An *irreflexive partial order* is a binary relation that is irreflexive, antisymmetric, and transitive.

The formal subject matter of this paper is concerned with finite and infinite sequences describing the executions of automata. Usually, we are discussing sequences of elements from a universal set of *actions*. Formally, a *sequence* β of actions is a mapping from a prefix of the positive integers to the

set of actions. We describe the sequence by listing the images of successive integers under the mapping, writing $\beta = \pi_1\pi_2\pi_3 \cdots$.[2] Since the same action may occur several times in a sequence, it is convenient to distinguish the different occurrences. Thus, we refer to a particular occurrence of an action in a sequence as an *event*. Formally, an *event* in a sequence $\beta = \pi_1\pi_2 \cdots$ of actions is an ordered pair (i, π), where i is a positive integer and π is an action, such that π_i, the ith action in β, is π.

A set of sequences P is *prefix-closed* provided that whenever $\beta \in$ P and γ is a prefix of β, it is also the case that $\gamma \in$ P. Similarly, a set of sequences P is *limit-closed* provided that any sequence all of whose finite prefixes are in P is also in P. We refer to any nonempty, prefix-closed, and limit-closed set of sequences as a *safety property*.

2.1. THE INPUT/OUTPUT AUTOMATON MODEL. In order to reason carefully about complex concurrent systems such as those that implement atomic transactions, it is important to have a simple and clearly defined formal model for concurrent computation. The model we use for our work is the *input/output automaton* model [15, 16]. This model allows careful and readable descriptions of concurrent algorithms and of the correctness conditions that they are supposed to satisfy. The model can serve as the basis for rigorous proofs that particular algorithms satisfy particular correctness conditions.

This subsection contains an introduction to a special case of the model that is sufficient for use in this paper. In particular, in this paper we consider properties of finite executions only, and do not consider liveness or fairness properties.

Each system component is modeled as an I/O automaton, which is a mathematical object somewhat like a traditional finite-state automaton. However, an I/O automaton need not be finite-state, but can have an infinite-state set. The actions of an I/O automaton are classified as either input, output, or internal. This classification is a reflection of a distinction between events (such as the receipt of a message) that are caused by the environment, events (such as sending a message) that the component can perform when it chooses and that affect the environment, and events (such as changing the value of a local variable) that a component can perform when it chooses, but that are undetectable by the environment except through their effects on later events. In the model, an automaton generates output and internal actions autonomously, and transmits output actions instantaneously to its environment. In contrast, the automaton's input is generated by the environment and transmitted instantaneously to the automaton. The distinction between input and other actions is fundamental, based on who determines when the action is performed: an automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

2.1.1. *Action Signatures.* The formal description of an automaton's actions and their classification into inputs, outputs, and internal actions is given by its action signature. An *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write *in*(S), *out*(S), and *int*(S) for the three components of S, and refer to the actions in the three sets as the *input actions*,

[2] We use the symbols $\beta, \gamma, \ldots$ for sequences of actions and the symbols π, ϕ, and ψ for individual actions.

output actions, and *internal actions* of S, respectively. We let *ext*(S) = in(S) ∪ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local*(S) = int(S) ∪ out(S), and refer to the actions in local(S) as the *locally controlled actions* of S. Finally, we let *acts*(S) = in(S) ∪ out(S) ∪ int(S), and refer to the actions in acts(S) as the *actions* of S.

An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature *extsig*(S) = (in(S), out(S), ∅), that is, the action signature that is obtained from S by removing the internal actions.

2.1.2. *Input/Output Automata.* An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of four components:[3]

—an action signature *sig*(A),
—a set *states*(A) of *states*,
—a nonempty set *start*(A) ⊆ states(A) of *start states*,
—a transition relation *steps*(A) ⊆ states(A) × acts(sig(A)) × states(A), with the property that for every state s′ and input action π there is a transition (s′, π, s) in steps(A).

Note that the set of states need not be finite. We refer to an element (s′, π, s) of steps(A) as a *step* of A. The step (s′, π, s) is called an *input step* of A if π is an input action, and *output steps*, *internal steps*, *external steps*, and *locally controlled steps* are defined analogously. If (s′, π, s) is a step of A, then π is said to be *enabled* in s′. Since every input action is enabled in every state, automata are said to be *input-enabled.* The input-enabling property means that an automaton is not able to block input actions. If A is an automaton, we sometimes write *acts*(A) as shorthand for acts(sig(A)), and likewise for in(A), out(A), etc. An I/O automaton A is said to be *closed* if all its actions are locally controlled, that is, if in(A) = ∅.

Note that an I/O automaton can be nondeterministic, by which we mean two things: that more than one locally controlled action can be enabled in the same state, and that the same action, applied in the same state, can lead to different successor states. This nondeterminism is an important part of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general, since many results about nondeterministic algorithms apply a fortiori to all algorithms obtained by restricting the nondeterministic choices. Moreover, the use of nondeterminism helps to avoid cluttering algorithm descriptions and proofs with inessential details. Finally, the uncertainties introduced by asynchrony make nondeterminism an intrinsic property of real concurrent systems, and so an important property to capture in our formal model of such systems.

2.1.3. *Executions, Schedules, and Behaviors.* When a system is modeled by an I/O automaton, each possible run of the system is modeled by an "execution," an alternating sequence of states and actions. The possible activity of the system is captured by the set of all possible executions that can be generated by the automaton. However, not all the information contained in an execution is

[3] I/O automata, as defined in [15], also include a fifth component, which is used for describing fair executions. We omit it here as it is not needed for the results described in this paper.

important to a user of the system, nor to an environment in which the system is placed. We believe that what is important about the activity of a system is the externally visible events, and not the states or internal events. Thus, we focus on the automaton's "behaviors"—the subsequences of its executions consisting of external (i.e., input and output) actions. We regard a system as suitable for a purpose if any possible sequence of externally visible events has appropriate characteristics. Thus, in the model, we formulate correctness conditions for an I/O automaton in terms of properties of the automaton's behaviors.

Formally, an *execution fragment* of A is a finite sequence $s_0\pi_1s_1\pi_2 \cdots \pi_ns_n$ or infinite sequence $s_0\pi_1s_1\pi_2 \cdots \pi_ns_n \cdots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i for which s_{i+1} exists. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by *execs*(A), and the set of finite executions of A by *finexecs*(A). A state is said to be *reachable* in A if it is the final state of a finite execution of A.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched*(α). We say that β is a *schedule* of A if β is the schedule of an execution of A. We denote the set of schedules of A by *scheds*(A) and the set of finite schedules of A by *finscheds*(A).

The *behavior* of a sequence β of actions in acts(A), denoted by *beh*(β), is the subsequence of β consisting of actions in ext(A). The *behavior* of an execution fragment α of A, denoted by *beh*(α), is defined to be beh(sched(α)). We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by *behs*(A) and the set of finite behaviors of A by *finbehs*(A).

We say that a finite schedule β of A *can leave* A *in* state s if there is some finite execution α of A with final state s and with sched(α) = β. Similarly, a finite behavior β of A *can leave* A *in* state s if there is some finite execution α of A with final state s and with beh(α) = β. An *extended step* of an automaton A is a triple of the form (s', β, s), where s' and s are in states(A), β is a finite sequence of actions in acts(A), and there is an execution fragment of A having s' as its first state, s as its last state, and β as its schedule.

If β is any sequence of actions and Φ is a set of actions, we write $\beta|\Phi$ to denote the subsequence of β containing all occurrences of actions in Φ. If A is an automaton, we write $\beta|A$ for $\beta|$acts(A).

2.2. COMPOSITION. Often, a single system can also be viewed as a combination of several component systems interacting with one another. To reflect this in our model, we define a "composition" operation by which several I/O automata can be combined to yield a single I/O automaton. Our composition operator connects each output action of the component automata with the identically named input actions of any number (usually one) of the other component automata. In the resulting system, an output action is generated autonomously by one component and is thought of as being instantaneously transmitted to all components having the same action as an input. All such components are passive recipients of the input, and take steps simultaneously with the output step.

2.2.1. *Composition of Action Signatures.* We first define composition of action signatures. Let I be an index set that is at most countable. A collection

$\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible*[4] if the following properties hold:

(1) $\text{out}(S_i) \cap \text{out}(S_j) = \emptyset$ for all $i, j \in I$ such that $i \neq j$,
(2) $\text{int}(S_i) \cap \text{acts}(S_j) = \emptyset$ for all $i, j \in I$ such that $i \neq j$,
(3) no action is in $\text{acts}(S_i)$ for infinitely many i.

Thus, no action is an output of more than one signature in the collection, and internal actions of any signature do not appear in any other signature in the collection. Moreover, we do not permit actions involving infinitely many component signatures.

The *composition* $S = \prod_{i \in I} S_i$ of a collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

—$\text{in}(S) = \bigcup_{i \in I} \text{in}(S_i) - \bigcup_{i \in I} \text{out}(S_i)$,
—$\text{out}(S) = \bigcup_{i \in I} \text{out}(S_i)$,
—$\text{int}(S) = \bigcup_{i \in I} \text{int}(S_i)$.

Thus, output actions are those that are outputs of any of the component signatures, and similarly for internal actions. Input actions are any actions that are inputs to any of the component signatures, but outputs of no component signature.

2.2.2. *Composition of Automata.* A collection $\{A_i\}_{i \in I}$ of automata is said to be *strongly compatible* if their action signatures are strongly compatible. The *composition* $A = \prod_{i \in I} A_i$ of a strongly compatible collection of automata $\{A_i\}_{i \in I}$ has the following components:[5]

—$\text{sig}(A) = \prod_{i \in I} \text{sig}(A_i)$,
—$\text{states}(A) = \prod_{i \in I} \text{states}(A_i)$,
—$\text{start}(A) = \prod_{i \in I} \text{start}(A_i)$,
—steps(A) is the set of triples (s', π, s) such that for all $i \in I$, (a) if $\pi \in \text{acts}(A_i)$, then $(s'[i], \pi, s[i]) \in \text{steps}(A_i)$, and (b) if $\pi \notin \text{acts}(A_i)$, then $s'[i] = s[i]$.[6]

Since the automata A_i are input-enabled, so is their composition, and hence their composition is an automaton. Each step of the composition automaton involves all the automata that have a particular action in their action signature performing that action concurrently, while the automata that do not have that action in their signature do nothing. We often refer to an automaton formed by composition as a "system" of automata.

If $\alpha = s_0 \pi_1 s_1 \cdots$ is an execution of A, let $\alpha | A_i$ be the sequence obtained by deleting $\pi_j s_j$ when π_j is not an action of A_i, and replacing the remaining s_j by $s_j[i]$. Recall that we have previously defined a projection operator for action sequences. The two projection operators are related in the obvious way: $\text{sched}(\alpha | A_i) = \text{sched}(\alpha) | A_i$, and similarly $\text{beh}(\alpha | A_i) = \text{beh}(\alpha) | A_i$.

In the course of our discussions, we often reason about automata without specifying their internal actions. To avoid tedious arguments about compatibil-

[4] A weaker notion called "compatibility" is defined in [15], consisting of the first two of the three given properties only. For the purposes of this paper, only the stronger notion will be required.
[5] Note that the second and third components listed are just ordinary Cartesian products, while the first component uses the previous definition of composition of action signatures.
[6] We use the notation $s[i]$ to denote the ith component of the state vector s.

ity, henceforth, we assume that unspecified internal actions of any automaton are unique to that automaton, and do not occur as internal or external actions of any of the other automata we discuss.

All of the systems that we use for modeling transactions are closed systems, that is, each action is an output of some component. Also, each output of a component will be an input of at most one other component.

2.2.3. *Properties of Systems of Automata.* Here we give basic results relating executions, schedules, and behaviors of a system of automata to those of the automata being composed. The first result says that the projections of executions of a system onto the components are executions of the components, and similarly for schedules, etc.

PROPOSITION 1. *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \prod_{i \in I} A_i$. If $\alpha \in execs(A)$, then $\alpha | A_i \in execs(A_i)$ for all $i \in I$. Moreover, the same result holds for finexecs, scheds, finscheds, behs, and finbehs in place of execs.*

Converses can also be proved for all the parts of the preceding proposition. The following are most useful. They say that schedules and behaviors of component automata can be "patched together" to form schedules and behaviors of the composition.

PROPOSITION 2. *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \prod_{i \in I} A_i$.*

(1) *Let β be a sequence of actions in $acts(A)$. If $\beta | A_i \in scheds(A_i)$ for all $i \in I$, then $\beta \in scheds(A)$.*
(2) *Let β be a finite sequence of actions in $acts(A)$. If $\beta | A_i \in finscheds(A_i)$ for all $i \in I$, then $\beta \in finscheds(A)$.*
(3) *Let β be a sequence of actions in $ext(A)$. If $\beta | A_i \in behs(A_i)$ for all $i \in I$, then $\beta \in behs(A)$.*
(4) *Let β be a finite sequence of actions in $ext(A)$. If $\beta | A_i \in finbehs(A_i)$ for all $i \in I$, then $\beta \in finbehs(A)$.*

The preceding proposition is useful in proving that a sequence of actions is a behavior of a system A: It suffices to show that the sequence's projections are behaviors of the components of A and then to appeal to Proposition 2.

2.3. IMPLEMENTATION. We define a notion of "implementation" of one automaton by another. Let A and B be automata with the same external action signature, that is, with extsig(A) = extsig(B). Then A is said to *implement* B if finbehs(A) $\subseteq$ finbehs(B). One way in which this notion can be used is the following: Suppose we can show that an automaton B is correct, in the sense that its finite behaviors all satisfy some specified property. Then, if another automaton A implements B, A is also correct. One can also show that, if A implements B, then replacing B by A in any system yields a new system in which all finite behaviors are behaviors of the original system.

The definition of an implementation of B by A does not require that A exhibit every possible behavior of B. Rather, B is viewed as describing the acceptable behaviors, and the behaviors of A are constrained to be acceptable. This constraint is easy to satisfy: A could simply never produce any outputs. (Notice that A still has behaviors, since all automata are required to be

input-enabled.) In other words, there is no requirement that A actually do something. Such requirements take the form of liveness, which we do not address in this paper. They can be handled within the I/O automaton model using the parts of the model that deal with liveness.

One useful technique for showing that one automaton implements another is to give a correspondence between states of the two automata. Such a correspondence can often be expressed in the form of a kind of abstraction mapping that we call a possibilities mapping, defined as follows: Suppose A and B are automata with the same external action signature, and suppose f is a mapping from states(A) to the power set of states(B). That is, if s is a state of A, f(s) is a set of states of B. The mapping f is said to be a *possibilities mapping* from A to B if the following conditions hold:

(1) For every start state s_0 of A, there is a start state t_0 of B such that $t_0 \in f(s_0)$.
(2) Let s' be a reachable state of A, $t' \in f(s')$ a reachable state of B, and (s', π, s) a step of A. Then there is an extended step (t', γ, t) of B (possibly having an empty schedule) such that the following conditions are satisfied:
   (a) $\gamma|\text{ext}(B) = \pi|\text{ext}(A)$,
   (b) $t \in f(s)$.

The following proposition shows that giving a possibilities mapping from A to B is sufficient to show that A implements B.

PROPOSITION 3. *Suppose that A and B are automata with the same external action signature and there is a possibilities mapping from A to B. Then A implements B.*

2.4. PRESERVING PROPERTIES. Although automata in our model are unable to block input actions, it is often convenient to restrict attention to those behaviors in which the environment provides inputs in a sensible way, that is, where the environment obeys certain *well-formedness* restrictions. A useful way of discussing such restrictions is in terms of the notion that an automaton preserves a property of behaviors: As long as the environment does not violate the property, neither does the automaton. Such a notion is primarily interesting for safety properties. Let Φ be a set of actions and P a safety property for sequences of actions in Φ. Let A be an automaton with $\Phi \cap \text{int}(A) = \emptyset$. We say that A *preserves* P if $\beta\pi|\Phi \in P$ whenever $\beta|\Phi \in P$, $\pi \in \text{out}(A)$, and $\beta\pi|A \in \text{finbehs}(A)$.

Thus, if an automaton preserves a property P, the automaton is not the first to violate P: as long as the environment only provides inputs such that the cumulative behavior satisfies P, the automaton will only perform outputs such that the cumulative behavior satisfies P. In many cases of interest, we have $\Phi \subseteq \text{ext}(A)$; note that even in this case, the fact that an automaton A preserves P does not imply that all of A's behaviors, when restricted to Φ, satisfy P. It is possible for a behavior of A to fail to satisfy P if an input causes a violation of P. However, the following proposition gives a way to deduce that all of a system's behaviors satisfy P. The proposition says that, under certain conditions, if all components of a system preserve P, then all the behaviors of the composition satisfy P.

PROPOSITION 4. *Let* $\{A_i\}_{i \in I}$ *be a strongly compatible collection of automata and let* $A = \prod_{i \in I} A_i$. *Let* Φ *be a set of actions such that* $\Phi \cap int(A) = \emptyset$, *and let* P *be a safety property for actions in* Φ. *Suppose that for each* $i \in I$, A_i *preserves* P. *Then* A *preserves* P. *Furthermore, if* $\Phi \cap in(A) = \emptyset$, *then* $behs(A)|\Phi \subseteq P$. *That is, if* $\beta \in behs(A)$, *then* $\beta|\Phi \in P$.

PROOF. Let β be a sequence of actions such that $\beta|\Phi \in P$, $\pi \in out(A)$, and $\beta\pi|A \in finbehs(A)$. Then $\pi \in out(A_i)$ for some $i \in I$, and $\beta\pi|A_i \in finbehs(A_i)$, by Proposition 2. Since A_i preserves P, $\beta\pi|\Phi \in P$.

Now suppose that $\Phi \cap in(A) = \emptyset$, and let $\beta \in behs(A)$. Since A preserves P, by a simple induction, every finite prefix of $\beta|\Phi$ is in P. Then, $\beta|\Phi \in P$, by the limit-closure of P. □

3. *Basic Systems*

In this section, we define *basic systems*, the class of transaction-processing systems to which our results apply. Basic systems generalize both the generic systems of [4] and the pseudotime systems of [2]. We also define correctness conditions for basic systems, in particular, the notion of correct management of orphans.

3.1. OVERVIEW. Transaction-processing systems consist of user-provided transaction code, plus transaction-processing algorithms designed to coordinate the activities of different transactions. The transactions are written by application programmers in a suitable programming language. Transactions are permitted to invoke operations on data objects. In addition, if nesting is allowed, transactions can invoke subtransactions and receive responses from the subtransactions describing the results of their processing.

In a transaction-processing system, the transaction-processing algorithms interact with the transactions, making decisions about when to schedule subtransactions and operations on data objects. The transaction-processing algorithms include concurrency control and recovery algorithms. In many interesting cases (e.g., for locking algorithms), the transaction-processing algorithms can be naturally divided into a controller and a collection of objects, where each object includes concurrency control and recovery algorithms appropriate for that object and the controller manages communication among the transactions and objects. We do not, however, require this division for our general results.

The transaction-processing systems studied in this paper are called *basic systems*. In the organization we consider, the transaction-processing algorithms are represented by a component called a *basic database*. Each component of a basic system is modeled as an I/O automaton. That is, each transaction is an automaton, and the basic database is another automaton.

The nested structure of transactions is modeled by describing each transaction and subtransaction in the transaction nesting structure as a separate I/O automaton. If a parent transaction T wishes to invoke a child transaction T′, T issues an output action that "requests that T′ be created." The basic database receives this request, and at some later time might issue an action that is an input to the child T′ and corresponds to the creation of T′. Thus, the different transactions in the nesting structure comprise a forest of automata, communicating with each other indirectly through the basic database. The highest-level

transactions, that is, those that are not subtransactions of any other transactions, are the roots in this forest.

It is actually more convenient to model the transaction nesting structure as a tree rather than as a forest. Therefore, we add an extra root automaton as a dummy transaction, located at the top of the transaction nesting structure. The highest-level user-defined transactions are considered to be children of this new root. The root can be thought of as modeling the outside world, from which invocations of top-level transactions originate and to which reports about the results of such transactions are sent.

In the rest of this section, we define basic systems and state the correctness conditions that they are supposed to satisfy.

3.2. SYSTEM TYPES. We begin by defining a type structure that will be used to name the transactions and objects in a basic system.

A *system type* consists of the following:

—a set T of *transaction names*,
—a distinguished transaction name $T_0 \in T$,
—a subset *accesses* of T not containing T_0,
—a mapping *parent*: $T - \{T_0\} \rightarrow T$, which configures the set of transaction names into a tree, with T_0 as the root and the accesses as the leaves,
—a set X of *object names*,
—a mapping *object*: accesses $\rightarrow X$,
—a set V of *return values*.

Each element of the set *accesses* is called an *access* transaction name, or simply an *access*. Also, if object(T) = X, we say that T is an *access* to X.

In referring to the transaction tree, we use standard tree terminology, such as leaf node, internal node, child, ancestor, and descendant. As a special case, we consider any node to be its own ancestor and its own descendant, that is, the ancestor and descendant relations are reflexive. We also use the notion of a least common ancestor of two nodes.

The transaction tree describes the nesting structure for transaction names, with T_0 as the name of the dummy root transaction. Each child node in this tree represents the name of a subtransaction of the transaction named by its parent. The children of T_0 represent names of the top-level user-defined transactions. The accesses represent names for the lowest-level transactions in the transaction nesting structure; we use these lowest-level transaction names to model operations on data objects. Thus, the only transactions that actually access data are the leaves of the transaction tree, and these do nothing else. The internal nodes model transactions whose function is to create and manage subtransactions including accesses, but they do not access data directly.

The tree structure should 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 with infinite branching.

The set X is the set of names for the objects used in the system. Each access transaction name is assumed to be an access to some particular object, as designated by the *object* mapping. The set V of return values is the set of

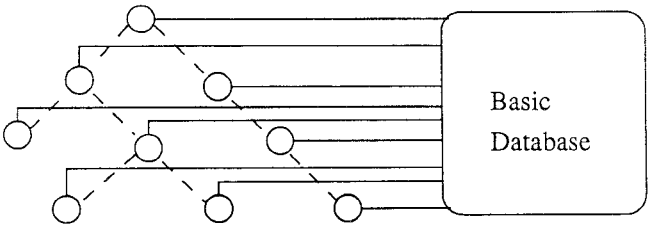

FIG. 1. Basic system.

possible values that might be returned by successfully completed transactions to their parent transactions.

For the rest of this paper, we fix a particular system type.

3.3. GENERAL STRUCTURE OF BASIC SYSTEMS. A basic system for a given system type is a closed system consisting of a *transaction automaton* A_T for each nonaccess transaction name T and a single *basic database automaton* B. Later in this section, we give conditions to be satisfied by the transaction and basic database automata. Here, we just describe the signatures of these automata, in order to explain how the automata are interconnected.

Figure 1 depicts the structure of a basic system.

The transaction nesting structure is indicated in part by dotted lines between transaction automata corresponding to parent and child. Access transactions do not have associated automata, so the diagram does not indicate the parents of accesses. The direct connections between automata (via shared actions) are indicated by solid lines. Thus, the transaction automata interact directly with the basic database, but not directly with each other.

Figure 2 shows the interface of a transaction automaton in more detail. The automaton for transaction name T has an input action CREATE(T), which is generated by the basic database in order to initiate T's processing. We do not include explicit arguments to a transaction in our model; rather we suppose that there is a different transaction for each possible set of arguments, so any input to the transaction is encoded in the name of the transaction. T has REQUEST_CREATE(T′) output actions for each child T′ of T in the transaction nesting structure; these are requests for creation of child transactions, and are communicated directly to the basic database. At some later time, the basic database might respond to a REQUEST_CREATE(T′) action by issuing a CREATE(T′) action; in case T′ is not an access, this action is an input to the automaton for transaction T′. T also has REPORT_COMMIT(T′, v) and REPORT_ABORT(T′) input actions, by which the basic database informs T about the fate (commit or abort) of its previously requested child T′. In the case of a commit, the report includes a return value v that provides information about the activity of T′; in the case of an abort, no information is returned. Finally, T has a REQUEST_COMMIT(T, v) output action, by which it announces to the basic database that it has completed its activity successfully, with a particular result that is described by return value v.

Figure 3 shows the basic database interface. The basic database in any particular basic system receives the previously mentioned REQUEST_CREATE and REQUEST_COMMIT actions as inputs from the transaction

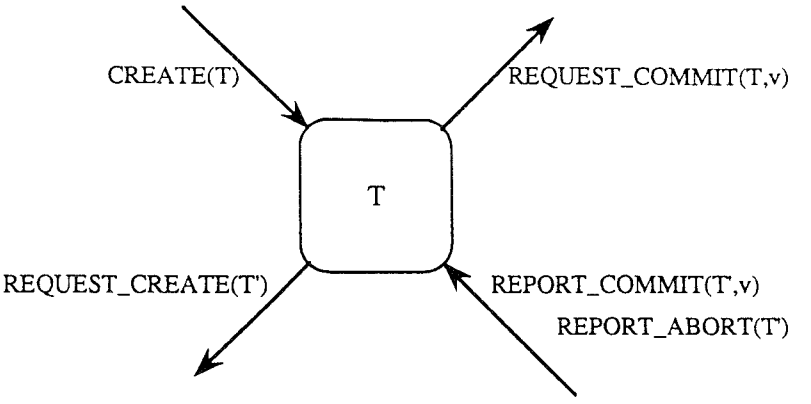

FIG. 2. Transaction interface.

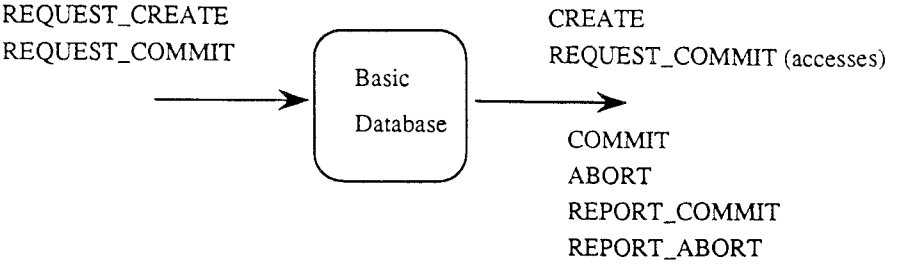

FIG. 3. Basic database interface.

automata. It produces CREATE actions as outputs, thereby awakening transaction automata or invoking operations on objects. The basic database also produces REQUEST_COMMIT(T, v) output actions for accesses T; these represent responses to the invocations of operations on objects. The value v in a REQUEST_COMMIT(T, v) action is a return value returned by the operation as part of its response. The basic database also produces COMMIT(T) and ABORT(T) actions for arbitrary transaction names $T \neq T_0$, representing decisions about whether the designated transaction commits or aborts. For technical convenience, we classify the COMMIT and ABORT actions and the REQUEST_COMMIT and CREATE actions for access transactions as output actions of the basic database, even when they are not inputs to any other system component.[7] The basic database also has REPORT_COMMIT and REPORT_ABORT actions as outputs, by which it communicates the fates of transactions to their parents.

Different basic databases may include additional output actions. The final section of this paper describes generic databases, which have additional out-

[7] Classifying actions as outputs even though they are not inputs to any other system component is permissible in the I/O automaton model. In this case, it would also be possible to classify these actions as internal actions of the basic database, but then the statements and proofs of the ensuing results would be more complicated.

puts by which the fates of transactions are communicated to the objects, so that locks may be released. Pseudotime databases are a second example, which contain additional outputs involving the management of timestamp data.

As is always the case for I/O automata, the components of a system are determined statically. Even though we referred earlier to the action of "creating" a child transaction, the model treats the child transaction as if it had been there all along. The CREATE action is treated formally as an input action to the child transaction; the child transaction will be constrained not to perform any output actions until such a CREATE action occurs. A consequence of this method of modeling dynamic creation of transactions is that the system must include automata for all possible transactions that might ever be created in any execution. In most interesting cases, this means that the system will include infinitely many transaction automata.

In our work, it is convenient to use two separate actions, REQUEST_CREATE and CREATE, to describe what happens when a subtransaction is activated. This separation occurs in actual distributed systems such as Argus, and is important in our results and proofs. Similar remarks hold for the distinction among REQUEST_COMMIT, COMMIT, and REPORT_COMMIT actions.

3.4. SERIAL ACTIONS. The external actions of a basic system of a given system type include the *serial actions* for that type. The *serial actions* for a given system type are defined to be the actions listed in the preceding subsection: CREATE(T) and REQUEST_COMMIT(T, v), where T is any transaction name and v is a return value, and REQUEST_CREATE(T), COMMIT(T), ABORT(T), REPORT_COMMIT(T, v), and REPORT_ABORT(T), where $T \neq T_0$ is a transaction name and v is a return value.[8] If β is a sequence of actions, define *serial*(β) to be the subsequence of β containing all the serial actions in β.

In this subsection, we define some simple concepts involving serial actions. All the definitions in this subsection are based on the set of actions only, and not on the specific automata in any particular system. For this reason, we present these definitions here, before going on (in the next subsection) to give more information about the basic system components.

We first present some fundamental definitions, and then we define notions of well-formedness for sequences of actions.

3.4.1. *Terminology.* The COMMIT(T) and ABORT(T) actions are called *completion* actions for T, while the REPORT_COMMIT(T, v) and REPORT_ABORT(T) actions are called *report* actions for T.

We associate transaction names with some of the serial actions, as follows. Let T be a transaction name. If π is either a CREATE(T) or a REQUEST_COMMIT(T, v) action, or is a REQUEST_CREATE(T'), REPORT_COMMIT(T', v') or REPORT_ABORT(T'), where T' is a child of T, then we define *transaction*(π) to be T. If π is a completion action, then transaction (π) is undefined. In some contexts, we also need to associate a transaction with

[8] These actions are called *serial actions* because they are exactly the external actions of a *serial system* of the given type. More will be said about serial systems later in the paper.

completion actions; since a completion action for T can be thought of as occurring in between T and parent(T), some of the time we want to associate T with the action, and at other times we want to associate parent(T) with it. Thus, we extend the transaction(π) definition in two different ways. If π is any serial action, then we define *hightransaction*(π) to be transaction(π) if π is not a completion action, and to be parent(T), if π is a completion action for T. Also, if π is any serial action, we define *lowtransaction*(π) to be transaction(π) if π is not a completion action, and to be T, if π is a completion action for T. In particular, hightransaction(π) = lowtransaction(π) = transaction(π) for all serial actions π for which transaction(π) is defined.

We also require notation for the object associated with any serial action whose transaction is an access. If π is a serial action of the form CREATE(T) or REQUEST_COMMIT(T,v), where T is an access to X, then we define *object*(π) to be X.

We extend the preceding notation to events as well as actions. For example, if π is an event, then we write *transaction*(π) to denote the transaction of the action of which π is an occurrence. We extend the definitions of hightransaction, lowtransaction, and object similarly. We extend other notation in this paper in the same way, without further explanation.

Now we require terminology to describe the status of a transaction during execution. Let β be a sequence of actions. A transaction name T is said to be *active* in β provided that β contains a CREATE(T) event but no REQUEST_COMMIT event for T. Similarly, T is said to be *live* in β provided that β contains a CREATE(T) event but no completion event for T. (However, note that β may contain a REQUEST_COMMIT for T.) Also, T is said to be an *orphan* in β if there is an ABORT(U) action in β for some ancestor U of T.

We have already used projection operators to restrict action sequences to particular sets of actions, and to actions of particular automata. We now introduce another projection operator, this time to sets of transaction names. Namely, if β is a sequence of actions and U is a set of transaction names, then $\beta|U$ is defined to be the sequence $\beta|\{\pi$: transaction(π) $\in U\}$. If T is a transaction name, we sometimes write $\beta|$T as shorthand for $\beta|\{$T$\}$. Similarly, if β is a sequence of actions and X is an object name, we sometimes write $\beta|$X to denote $\beta|\{\pi$: object(π) = X$\}$.

3.4.2. *Well-Formedness.* We place very few constraints on the transaction automata and basic database automaton in our definition of a basic system. However, we want to assume that certain simple properties are guaranteed; for example, a transaction should not take steps until it has been created, and the basic database should not create a transaction that has not been requested. Such requirements are captured by well-formedness conditions, which are fundamental safety properties of sequences of external actions of the transaction and basic database automata. We define those conditions here.

First, we define transaction well-formedness. Let T be any transaction name. A sequence β of serial actions π with transaction(π) = T is defined to be *transaction well-formed* for T provided the following conditions hold:

(1) The first event in β, if any, is a CREATE(T) event, and there are no other CREATE events,

(2) There is at most one REQUEST_CREATE(T′) event in β for each child T′ of T,
(3) Any report event for a child T′ of T is preceded by REQUEST_CREATE(T′) in β,
(4) There is at most one report event in β for each child T′ of T,
(5) If a REQUEST_COMMIT event for T occurs in β, then it is preceded by a report event for each child T′ of T for which there is a REQUEST_CREATE(T′) in β,
(6) If a REQUEST_COMMIT event for T occurs in β, then it is the last event in β.

In particular, if T is an access, then the only sequences that are transaction well-formed for T are the prefixes of the two-event sequences of the form CREATE(T)REQUEST_COMMIT(T, v). For any T, it is easy to see that the set of transaction well-formed sequences for T is a safety property, that is, that it is prefix-closed and limit-closed.

Next, we define basic database well-formedness. A sequence β of serial actions is defined to be *basic database well-formed* provided the following conditions hold:

(1) The sequence $\beta|T$ is transaction well-formed, for all transaction names T,
(2) If a CREATE(T) event occurs in β, for $T \neq T_0$, then there is a preceding REQUEST_CREATE(T) in β,
(3) If there is a COMMIT(T) event in β, then there is a preceding REQUEST_COMMIT(T, v) event in β, for some v,
(4) If there is an ABORT(T) event in β, then there is a preceding REQUEST_CREATE(T) event in β,
(5) There is at most one completion event in β for each transaction name T,
(6) If there is a REPORT_COMMIT(T, v) event in β, then there is a preceding REQUEST_COMMIT(T, v) event in β and a preceding COMMIT(T) event in β,
(7) If there is a REPORT_ABORT(T) event in β, then there is a preceding ABORT(T) event in β,
(8) There is at most one report event in β for each transaction name T.

3.5. BASIC SYSTEMS. We are now ready to define *basic systems*. Basic systems are composed of transaction automata, one for each nonaccess transaction name, and a single basic database automaton. We describe the two kinds of components in turn.

3.5.1. *Transaction Automata.* A *transaction automaton* A_T for a nonaccess transaction name T (of the given system type) is an I/O automaton with the following external action signature:

Input:
  CREATE(T)
  REPORT_COMMIT(T′, v), for every child T′ of T, and return value v
  REPORT_ABORT(T′), for every child T′ of T
Output:
  REQUEST_CREATE(T′), for every child T′ of T
  REQUEST_COMMIT(T, v), for every return value v

In addition, A_T may have an arbitrary set of internal actions. We require A_T to preserve transaction well-formedness for T, as defined in the preceding subsection. As discussed earlier, this does not mean that all behaviors of A_T are transaction well-formed, but it does mean that as long as the environment of A_T does not violate transaction well-formedness, A_T will not do so. Except for that requirement, transaction automata can be chosen arbitrarily. Note that if β is a sequence of actions, then $\beta|T = \beta|\text{ext}(A_T)$.

Transaction automata are intended to be general enough to model the transactions defined in any reasonable programming language. Particular programming languages may impose additional restrictions on transaction behavior. (For example, Argus suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

3.5.2. *Basic Database Automata.* A *basic database automaton* is also modeled as an I/O automaton. A basic database passes requests for the creation of subtransactions to the appropriate recipient, initiates REQUEST_COMMIT actions for accesses, makes decisions about the commit or abort of transactions, and passes reports about the completion of children back to their parents. It may also carry out other activity.

A basic database has the following actions in its external action signature:

Input:
REQUEST_CREATE(T), $T \neq T_0$
REQUEST_COMMIT(T, v), T a nonaccess transaction name
Output:
CREATE(T)
REQUEST_COMMIT(T, v), T an access transaction name
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T, v), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$

In addition, it may have other arbitrary output actions, as well as arbitrary internal actions. Depending upon the design of the particular basic database automaton, some of the additional output actions may be associated with particular objects. Hence, each basic database is assumed to come equipped with an extension of the *object* partial mapping on actions, which may associate some of these additional, nonserial output actions with particular object names. That is, each nonserial output action π may (but need not) have object(π) defined.

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction automata, and conversely, all the CREATE and report outputs (except those CREATE(T) actions for which T is an access) are identified with the corresponding inputs of transaction automata. A basic database is required to preserve basic database well-formedness.

There are many examples of basic databases in the literature. For example, the composition of the generic controller and generic objects of [4] preserves basic database well-formedness, and so is an example of a basic database. The same is true for the composition of the pseudotime controller and pseudotime

objects of [2]. We present these examples in more detail later in this paper. In fact, we claim that almost all interesting transaction-processing algorithms can be modeled as basic databases. (See [14] for additional examples.)

Our notion of basic database identifies the aspects of transaction-processing algorithms that are relevant to our analysis of orphan management algorithms. It turns out that the details of how synchronization and recovery are implemented by a basic database 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 management algorithms in a way that is independent of the concurrency control and recovery methods used within the basic database.

3.5.3. *Basic Systems.* A *basic system* B is the composition of a strongly compatible set of automata indexed by the union of the set of nonaccess transaction names and the singleton set {BD} (for "basic database"). Associated with each nonaccess transaction name T is a transaction automaton A_T for T. Associated with the name BD is a basic database automaton for the system type.

When the particular basic system B is understood from the context, we call its external actions the *basic actions*, and its executions, schedules, and behaviors the *basic executions*, *basic schedules*, and *basic behaviors*, respectively. The following proposition says that basic behaviors have the appropriate well-formedness properties:

PROPOSITION 5. *If β is a basic behavior, then the following conditions hold.*

(1) *For every transaction name T, $\beta|T$ is transaction well-formed for T.*
(2) *The sequence serial(β) is basic database well-formed.*

PROOF. Note first that the basic database preserves basic database well-formedness, and this immediately implies that it preserves transaction well-formedness for every transaction name. Next, note that each transaction automaton preserves transaction well-formedness for the appropriate transaction name. Furthermore, it has in its signature no actions of other transactions, and so preserves transaction well-formedness for all transaction names. The first part of the proposition follows by Proposition 4.

A simple induction shows that each transaction automaton also preserves basic database well-formedness, and the second conclusion follows also from Proposition 4. □

3.6. SERIAL CORRECTNESS. In this subsection, we give appropriate notions of correctness for basic systems. These include notions appropriate for systems that manage orphans, as well as notions for systems that do not manage orphans but do carry out concurrency control and recovery. These notions are taken from [4], [12], and [13]. The spirit of our definitions is similar to that of the usual definition of serializability in the database literature. However, the usual notion does not take nesting or aborts into account.

We define correctness conditions for basic systems of a given type by relating their behaviors to those of a particular basic system of that type, the serial system. The executions, schedules, and behaviors of a serial system are called serial executions, serial schedules, and serial behaviors, respectively. Serial systems are composed of transaction automata and a serial database, which

itself is the composition of a serial scheduler and objects. The transaction automata are identical to those in basic systems. The serial scheduler controls the order in which the transactions take steps and in which accesses to objects occur. It permits only one child of a transaction to run at a time. Thus, sibling transactions execute sequentially at every parent node in the transaction tree, so that transactions are run in a depth-first traversal of the tree. Also, the serial scheduler aborts a transaction only if it has not been created, and creates a transaction only if it has not been aborted. Thus, in a serial execution, sibling transactions execute sequentially and aborted transaction take no steps.

Objects in a serial system are quite simple. Since the serial scheduler 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 serve the same purpose as the *serial specifications* [25] and [26].) A detailed description of serial systems may be found in the references [4] and [12–14].

Now we give a definition that says that a sequence of actions "looks like" a serial behavior to a particular transaction. Namely, if β is a sequence of actions and T is a transaction name, we say that β is *serially correct for* T if there exists a serial behavior γ such that $\gamma|T = \beta|T$. In other words, T sees the same thing in β that it could see in some serial behavior.

Now we can define two notions of correctness for basic systems. First, we say that a basic system is *serially correct* if each of its finite[9] behaviors is serially correct for all transaction names.[10] The requirement that every transaction see a serial view is very strong. Without orphan management, in fact, systems may not meet this requirement. (This is true of all published concurrency control algorithms for nested transactions of which we are aware.) Instead, they provide a slightly weaker notion of correctness, namely that nonorphan transactions see serial views. More precisely, we say that a basic system is *serially correct for nonorphans* if each of its finite behaviors β is serially correct for all transaction names that are not orphans in β. Orphans, however, can see arbitrary views.

The papers [2], [4], [12], and [13] contain examples of basic systems that are serially correct for nonorphans. The basic system in [12] and [13] uses exclusive locking for concurrency control and recovery,[11] while the systems in [4] use a more general commutativity-based locking strategy. The systems in [2] use timestamps for concurrency control and recovery.

The orphan management algorithms of this paper ensure that the systems that use them are serially correct. To ensure this, the orphan management algorithms rely on the basic database to ensure serial correctness for nonorphans; in fact, the algorithms work with any basic database that ensures serial correctness for nonorphans. In this sense, the orphan management algorithms

[9] Serial correctness is stated in terms of finite behaviors because the corresponding property for infinite behaviors is not satisfied by locking algorithms, in the absence of extra assumptions [22].

[10] As discussed in [12], this definition of correctness allows different transactions in β to "see" different serial behaviors. However, correctness applies to the root transaction T_0 as well, so the root must see the same results from the top-level transactions that it could see in some serial behavior.

[11] There are some minor differences; for example, the completion and report actions are combined into single actions rather than treated as two separate actions.

and the concurrency control algorithms are independent. We prove a result of the following sort for each orphan management algorithm: If β is a behavior of the system with orphan management and T is a transaction name, then there exists a behavior γ of the underlying basic system such that $\gamma|T = \beta|T$ and T is not an orphan in γ. In other words, the orphan management algorithms prevent transactions from "knowing" that they are orphans—everything a transaction sees is consistent with what it could see in the underlying basic system in some execution in which it is not an orphan. These results imply that if the basic system is serially correct for nonorphans, then the corresponding system with orphan management is serially correct.

4. *Information Flow*

In this section, we define families of irreflexive partial orders, each of which models the information flow between events in behaviors of a basic system. We call these partial orders *affects relations*; if an event ϕ does not affect an event π in an execution, then π cannot "know" that ϕ occurred, in the sense that there is a "possible world" in which π occurs but ϕ does not. The algorithms described later in this paper ensure that no event of a transaction is affected by the abort of an ancestor; thus, no event of a transaction ever knows that the transaction is an orphan. To make our definitions as general as possible, we define affects relations by describing the constraints they must satisfy. Later in the paper, we give examples of affects relations for particular kinds of systems.

4.1. Families of Affects Relations. If β is a sequence of basic actions, R is a binary relation on events in β, and γ is a subsequence of β, then we say that γ is R-*closed* in β if, whenever γ contains an event π in β, it also contains any event ϕ in β such that $(\phi, \pi) \in R$.

Let B be a basic system, and let $R = \{R_\beta\}$ be a family of relations, one for each sequence β of external actions of B. Then R is said to be a *family of affects relations* for B provided that the following conditions hold.

(1) Each R_β is an irreflexive partial order on the events in β such that if $(\phi, \pi) \in R_\beta$ then ϕ precedes π in β,
(2) If γ is a prefix of β, and ϕ and π are in γ, then $(\phi, \pi) \in R_\beta$ if and only if $(\phi, \pi) \in R_\gamma$,
(3) If β is a behavior of B and γ is an R_β-closed subsequence of β, then γ is a behavior of B,
(4) Suppose that β is a behavior of B and $(\phi, \pi) \in R_\beta$, where ϕ is an ABORT(T′) event, and π is a CREATE, a COMMIT, an ABORT, a REPORT_COMMIT, a REPORT_ABORT(T) for $T \neq T'$, an output of a nonaccess transaction, or a REQUEST_COMMIT for an access. Then there is an event ψ between ϕ and π in β such that $(\phi, \psi) \in R_\beta$, where ψ is related to π as follows:
    (a) If π = CREATE(T), then ψ = REQUEST_CREATE(T),
    (b) If π = COMMIT(T), then ψ = REQUEST_COMMIT(T, v),
    (c) If π = REPORT_COMMIT(T, v), then ψ = COMMIT(T),
    (d) If π = REPORT_ABORT(T), then ψ = ABORT(T),
    (e) If π = ABORT(T), then ψ = REQUEST_CREATE(T),
    (f) If π is an output of nonaccess transaction T, then ψ is an event of transaction T,

(g) If π = REQUEST_COMMIT(T, v) where T is an access to object X, then object(ψ) = X.[12]

The first two conditions are quite simple; the first says that each relation R_β describes a partial ordering consistent with the order in which events appear in β, and the second says that whether or not one event affects another is determined at the time the second event occurs. The third condition describes a sense in which the relation R_β captures all the dependency relationships between events. The condition implies that if π is not affected by ϕ in some behavior β, then π cannot "know" that ϕ occurred, since π could also have occurred in a different behavior in which ϕ did not occur. The fourth condition is a technical condition that describes certain limitations on the pattern of information flow. It will be needed for our later proofs, and can be demonstrated for the examples in Section 9.

When the particular family $R = \{R_\beta\}$ of affects relations is understood, we often refer to each R_β as an *affects relation*, and we often say "ϕ affects π in β" to mean that $(\phi, \pi) \in R_\beta$.

4.2. FAMILIES OF DIRECTLY-AFFECTS RELATIONS. In many cases of interest, a family of affects relations can be conveniently described by a generating family of smaller relations. Thus, let B be a basic system and $R' = \{R'_\beta\}$ be a family of relations, one for each sequence β of external actions of B. Then R' is said to be a *family of directly-affects relations* for B, provided that there is a family $R = \{R_\beta\}$ of affects relations for B such that for each β, R_β is the transitive closure of R'_β. In this case, we say that R' *generates* R. Note that there is no requirement that the relations in R' be minimal.

When the particular family $R' = \{R'_\beta\}$ of directly-affects relations is understood, we often refer to each R'_β as a *directly-affects relation*; also, we often say that "ϕ directly affects π in β" to mean that $(\phi, \pi) \in R'_\beta$.

4.3. USING FAMILIES OF AFFECTS RELATIONS TO DESCRIBE ORPHAN MANAGEMENT ALGORITHMS. The intuitive idea behind the orphan management algorithms is that they ensure that an event of a transaction T is never affected by the abort of an ancestor. Then we can show that every transaction gets a view it could get in a behavior in which it is not an orphan; we simply take the subsequence of the original behavior containing all events of T and all events that affect them in that behavior. The resulting sequence is a basic behavior, by the definition of a family of affects relations, and does not contain an abort for an ancestor of T, by construction.

The following example illustrates how an orphan can see an inconsistent state in a system without orphan management. Suppose that T is a transaction with children T_1 and T_2, both of which are accesses to an object X. Consider the following scenario: T first accesses X through its child T_1. T then requests the creation of T_2, which will access X again. Furthermore, assume that T_2 is requested only if T_1 completes successfully, and that T_1 modifies X. Now suppose that T aborts before T_2 starts running at X, and X learns of the abort. If T_1's modification of X is undone when X learns that T aborted, then T_2 will not see the value for X that it expects, since T_2 only runs if T_1 has modified X successfully. This scenario is captured more precisely by the following fragment

[12] Note that ψ can be either a serial or a nonserial event.

of a schedule of a generic system (generic systems are described in more detail in Section 9.1):

CREATE(T)
REQUEST_CREATE(T_1)
CREATE(T_1)
REQUEST_COMMIT(T_1, v_1)
COMMIT(T_1)
REPORT_COMMIT(T_1, v_1)
REQUEST_CREATE(T_2)
ABORT(T)
INFORM_ABORT_AT(X)OF(T)
CREATE(T_2)
REQUEST_COMMIT(T_2, v_2)
⋮

(The INFORM_ABORT event lets X know that T has aborted.) The family of affects relations for generic systems described in Section 9.1 ensures that the ABORT(T) event affects the INFORM_ABORT_AT(X)OF(T) event, and that an event at an object is affected by all prior events at the object. Thus, by preventing T_2 from running when its events would be affected by the abort of an ancestor, we can prevent it from knowing that it is an orphan.

5. *Filtered Systems*

The two orphan management algorithms analyzed in this paper use quite different techniques. However, each can be proved correct by showing that it implements the same abstract algorithm, described in this section.

For Sections 5 through 8 of this paper, we fix a particular but arbitrary basic system B, together with a family R of affects relations for B and a family R′ of directly-affects relations for B, where R′ generates R. We describe several algorithms that exploit the properties of the affects relations to manage orphans. In Section 9, we illustrate this general development by describing two specific basic systems and their affects relations.

One way of ensuring that actions of a transaction T are never affected by the abort of an ancestor of T is to add preconditions to all the actions of the basic database to permit actions 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 REQUEST_COMMIT actions occur for access transactions. We then show that this is sufficient to ensure that transactions are never affected by the aborts of ancestors.

We construct a filtered system based on the given basic system B and the given family R of affects relations. The filtered system consists of the given transaction automata from B and a *filtered database automaton*. The filtered database automaton is obtained by slightly modifying the basic database automaton; it "filters" REQUEST_COMMIT actions of access transactions so that any transaction, orphan or not, sees a view it could see as a nonorphan in the basic system.

5.1. THE FILTERED DATABASE. The filtered database is obtained via a simple transformation from the basic database. The only difference between the behaviors of the two databases is that the new database only allows a REQUEST_COMMIT of an access to occur if it is not affected by the abort of an ancestor.

The filtered database has the same signature as the basic database. The state of the filtered database has two components, *basic_state* and *history*, where basic_state is a state of the basic database and history is a sequence of basic actions. Initial states of the filtered database are those with basic_state equal to an initial state of the basic database and history equal to the empty sequence.

A triple (s', π, s) is a step of the filtered database if and only if the following conditions hold:

(1) (s'.basic_state, π, s.basic_state) is a step of the basic database,
(2) s.history = s'.historyπ if π is a basic action,[13]
(3) s.history = s'.history if π is not a basic action,
(4) If π = REQUEST_COMMIT(T, v) where T is an access to object X, and if T' is an ancestor of T, then no ABORT(T') event affects an event ψ with object(ψ) = X in s'.history.

Thus, at the point where the REQUEST_COMMIT of an access is about to occur, an explicit test is performed to verify that no preceding event at the same object is affected by the abort of any ancestor of the access.

LEMMA 6. *Let β be a finite schedule of the filtered database that can leave the filtered database in state s. Then s.history = beh(β).*

5.2. THE FILTERED SYSTEM. The *filtered system* is the composition of the transaction automata and the filtered database automaton. We call its executions, schedules, and behaviors the *filtered executions*, *filtered schedules*, and *filtered behaviors*, respectively.

LEMMA 7. *The filtered system implements the basic system.*

PROOF. The mapping f that assigns to each state s of the filtered system the singleton set f(s) consisting of s.basic_state is easily seen to be a possibilities mapping. Proposition 3 implies the result. □

As described above, the filtered database performs an explicit test to ensure that the REQUEST_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 events.

LEMMA 8. *Let β be a filtered behavior and let T be any transaction name. Let ρ be an event in β such that transaction(ρ) = T. Then there is no ABORT(T') event ϕ such that $(\phi, \rho) \in R_\beta$, for any ancestor T' of T.*

PROOF. First note that Lemma 7 and Proposition 5 imply that serial(β) is basic database well-formed. The proof of the lemma is by induction on the length of β. If β is empty, the result clearly holds. Suppose $\beta = \beta'\pi$, and that the lemma holds for β'. From the restrictions on affects relations, $R_\beta \subseteq R_{\beta'} \cup$

[13] Recall that internal actions of the basic database are not classified as basic actions.

$\{(\phi, \pi) | \phi$ is an action in $\beta'\}$. Thus, by induction, it suffices to show that the lemma holds when $\rho = \pi$.

Suppose that the lemma does not hold, that is, that ϕ = ABORT(T′) affects π in β, where transaction(π) = T and T′ is an ancestor of T. We derive a contradiction. We consider cases.

(1) T *is a nonaccess and* π *is an output action of* T. Then, by the fourth property of affects relations, there is an event ψ of T between ϕ and π such that ϕ affects ψ in β'. This contradicts the inductive hypothesis.
(2) T *is an access to object* X *and* π *is a REQUEST_COMMIT for* T. Then, by the fourth property of affects relations, there is an event ψ of object X between ϕ and π in β' such that ϕ affects ψ in β'. Then, the precondition for π in the filtered database is violated, a contradiction.
(3) π *is CREATE*(T). Then, by the fourth property of affects relations, there is a REQUEST_CREATE(T) event ψ between ϕ and π such that ϕ affects ψ in β'. Since REQUEST_CREATE(T) is an action 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 β. But this implies that serial(β) is not basic database well-formed, a contradiction.
(4) T *is a nonaccess and* π *is REPORT_COMMIT*(T″, v), *where* T″ *is a child of* T. Then T′ is an ancestor of T″. By the fourth property of affects relations, there is a REQUEST_COMMIT(T″, v) event ψ between ϕ and π such that ϕ affects ψ in β'. Since transaction (ψ) = T″, this contradicts the inductive hypothesis.
(5) T *is a nonaccess and* π *is REPORT_ABORT*(T″), *where* T″ *is a child of* T. By the fourth property of affects relations, there is a REQUEST_CREATE(T″, v) event ψ between ϕ and π such that ϕ affects ψ in β'. Since transaction(ψ) = T, this contradicts the inductive hypothesis.
(6) π *is a nonserial basic action.* Then, transaction(π) is undefined, a contradiction. □

5.3. SIMULATION OF THE BASIC SYSTEM BY THE FILTERED SYSTEM. The following theorem is the key result of this paper. It shows that the filtered system ensures that every transaction gets a view it could get in the basic system when it is not an orphan. (Formally, a transaction T's "view" in a behavior β is its local behavior, $\beta|$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 management algorithms.

THEOREM 9. *Let* β *be a filtered behavior and let T be a transaction name. Then there exists a basic behavior* γ *such that T is not an orphan in* γ *and* $\gamma|T = \beta|T$.

PROOF. Let γ be the subsequence of β containing all actions π such that transaction(π) = T, and all other actions ϕ that affect, in β, some action whose transaction is T. Since R_β is a transitive relation, γ is R_β-closed in β. By the definition of a family of affects relations, γ is a basic behavior. It suffices to show that there is no ancestor T′ of T for which ABORT(T′) occurs in γ. Suppose not; that is, there exists an ancestor T′ of T for which

ABORT(T′) occurs in γ. Then by the construction of γ, β contains an event π of T such that an ABORT(T′) event ϕ affects π in β. By Lemma 8, this is impossible. □

We obtain an important corollary of Theorem 9.

COROLLARY 10. *If the basic system is serially correct for nonorphans, then the filtered system is serially correct.*[14]

PROOF. Let β be a filtered behavior and let T be any transaction name. Theorem 9 yields a behavior δ of the basic system such that T is not an orphan in δ and $\delta|T = \beta|T$. Since the basic system is serially correct for nonorphans, there is a serial behavior γ with $\gamma|T = \delta|T$; this is equal to $\beta|T$, as needed. □

At first, it might seem somewhat surprising that it is enough to prevent the REQUEST_COMMIT events of orphan accesses to ensure serial correctness for all orphans. The reason it is not necessary to filter other actions is because of the assumptions about affects relations. Essentially, these assumptions indicate that an event, say CREATE(T), cannot "know" about an ABORT(T′) event unless some earlier event, in this case REQUEST_CREATE(T), already "knows" about the ABORT(T′). (The assumption about affects relations that an affects-closed subsequence of a behavior is itself a behavior implies that if π is not affected by ϕ, π cannot know that ϕ has occurred, since there is a behavior of the system in which π occurs and ϕ does not.) If we assume inductively that REQUEST_CREATE(T) does not know about the abort of an ancestor of T, then the properties assumed for affects relations guarantee that CREATE(T) will not know about it either.

The assumptions about affects relations derive from the restricted communication patterns in typical systems: A nonaccess transaction receives information from its parent when it is created, and from its children when they report, but not from any other source. Access transactions may receive information from other accesses (e.g., accesses to the same object share state), but can only affect nonaccesses through a REPORT_COMMIT event, which must be preceded by a REQUEST_COMMIT. As long as T does not receive reports from any accesses that "know" that its ancestor has aborted, T cannot observe a state that depends on the abort. In effect, by preventing REQUEST_COMMIT actions for orphan accesses, we isolate orphan transactions from the objects, ensuring that an orphan transaction never sees that it is an orphan.

Sometimes we want to be explicit about the dependency of the filtered system on the particular basic system and family of affects relations from which it is derived. Thus, we restate the corollary above in a form that exhibits this dependency. If B is a basic system and R is a family of affects relations for B then let Filtered(B, R) be the corresponding filtered system; it is composed of the same transaction automata and the filtered database that corresponds to the given basic database.

[14] It should be easy to see that the filtered system is also a basic system, and so the notion of serial correctness has been defined for filtered systems. Similar comments hold for the rest of the systems described in this paper.

COROLLARY 11. *Let B be a basic system and R a family of affects relations for B. If B is serially correct for nonorphans, then Filtered(B, R) is serially correct.*

6. *Argus Systems*

In this section, we analyze the orphan management algorithm used in the Argus system [9, 10]. We describe the algorithm by defining an *Argus database* that describes in formal terms the algorithm discussed in [10]. As with the filtered database, the Argus database is obtained from the basic database via a simple construction. We then define the Argus system, which is composed of transactions and an Argus database, and show that the Argus system implements the filtered system. Thus, if the filtered system is serially correct, so is the corresponding Argus system.

6.1. THE ARGUS DATABASE.

The filtered database uses global knowledge of the entire history of actions to filter the REQUEST_COMMIT actions 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 REQUEST_COMMIT of an access is not affected by the abort of an ancestor, the Argus algorithm keeps track of the aborts "known" by each event that occurs, and propagates this knowledge from an event to any later events that it affects.

In the actual Argus system, knowledge about aborts is propagated in messages sent over the network; we model this formally by propagating knowledge from an event to every event that it *directly affects*. A poor choice of a directly-affects relation could make the algorithm we describe here hard to implement. However, if one event ϕ directly affects another event π only if the two events occur at the same site, or if a message is sent from ϕ's site to π's after ϕ occurs and before π occurs, then it is straightforward to implement the algorithm using information available locally at each site, by transmitting the information about aborted transactions on messages sent between sites.

The construction of the Argus database makes use of the given family R′ of directly-affects relations for B. The Argus database has the same signature as the basic database. The state of the Argus database has three components: *basic_state*, *history*, and *known_aborts*, where basic_state is a state of the basic database, history is a sequence of basic actions, and known_aborts is a partial mapping from basic events to sets of transactions. This mapping records the transactions whose aborts affect each event that has occurred. The set known_aborts(π) may actually include more transactions than those whose aborts affect π. By adding more aborted transactions to this set, an implementation would restrict the behavior of orphans further than is strictly necessary to ensure the correctness conditions. In Argus, for example, each event occurs at some physical node of the network, and each node manages a single set of aborted transactions for the entire set of events that occur at that node. In this case, known_aborts(π) includes at least the transactions whose aborts affect π, as well as those transactions whose aborts affect any other event that has occurred at the same physical node as π.

Initial states of the Argus database are those with basic_state equal to an initial state of the basic database, history equal to the empty sequence, and known_aborts everywhere undefined. A triple (s′, π, s) is a step of the Argus

database if and only if the following conditions hold:

(1) (s'.basic_state, π, s.basic_state) is a step of the basic database,
(2) s.history = s'.historyπ, if π is a basic action,
(3) s.history = s'.history if π is not a basic action,
(4) If π is a basic action and $(\phi, \pi) \in R'_{s.history}$ then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π),
(5) If $\phi \neq \pi$, then s.known_aborts(ϕ) = s'.known_aborts(ϕ),
(6) If π is a basic action and ϕ is an ABORT(T) event in s'.history such that $(\phi, \pi) \in R'_{s'.history}$, then T $\in$ s.known_aborts(π),
(7) If π is a REQUEST_COMMIT(T, v) action for an access T to object X, then there is no ancestor of T in s'.known_aborts(ϕ), for any event ϕ of object X in s'.history.

There are two significant differences between the Argus database and the basic database. First, the effects for each action π in the Argus database require s.known_aborts(π) to include s'.known_aborts(ϕ) for each ϕ that directly affects π. In addition, an event π that is directly affected by ABORT(T) requires T to be in known_aborts(π). As Lemma 16 below shows, these constraints are enough to ensure that s.known_aborts(π) contains T whenever ABORT(T) affects π.

Second, the precondition for the REQUEST_COMMIT of an access to X permits the event to occur only if the access does not "know about" the abort of an ancestor, that is, no ancestor is in s'.known_aborts(ϕ) for any event ϕ of object X in s'.history. As Lemma 17 below shows, this is enough to ensure that every Argus behavior is a filtered behavior.

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

The known_aborts component is managed so as to ensure 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 known_aborts mapping at a coarser granularity. (By maintaining the known_aborts 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 more than necessary.

Notice also that the Argus database does not put any upper limit on what goes into the known_aborts mapping. For example, it is permissible for known_aborts(π) to contain a transaction that has not aborted. This might cause nonorphans to block, but will otherwise not result in incorrect behavior. It would be easy (and intuitively appealing) to add a requirement that known_aborts(π) only includes aborted transactions, but this is not necessary for proving that the algorithm prevents all orphans from seeing inconsistent views. To prove other properties, such as that the algorithm only detects real orphans, we would need to add additional requirements such as the one just mentioned. We do 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 an appropriate subject of further research.

Finally, while the Argus database is distinguished from the filtered database by maintaining information about ABORT actions on a more local basis, we have kept the state component s.history, which maintains global knowledge of the past behavior of the system. A practical implementation of this algorithm would maintain less voluminous history information in a distributed fashion. Examination of the additional preconditions imposed by the Argus database on any action π reveals that s.history is used to determine the events ϕ that directly affect π, and when π is a REQUEST_COMMIT(T, v) action for an access T to object X, to determine the events that precede π at X. The details of efficient maintenance of sufficient history information are dependent on the particular basic database and distribution scheme, and are not addressed further in this paper.

6.2. THE ARGUS SYSTEM. The *Argus system* is the composition of transactions and the Argus database. Executions, schedules, and behaviors of the Argus system are called *Argus executions*, *Argus schedules*, and *Argus behaviors*, respectively.

LEMMA 12. *The Argus system implements the basic system.*

PROOF. The proof is similar to that of Lemma 7. □

LEMMA 13. *Let β be a finite Argus behavior that can leave the Argus database in state s. Then s.known_aborts(π) is defined if and only if π is an event in β.*

The following lemma says that each known_aborts set is defined at most once during an Argus execution.

LEMMA 14. *Let $\beta'\beta$ be a finite Argus schedule, where β' can leave the Argus database in state s' and (s', β, s) is an extended step of the Argus database. If s'.known_aborts(π) is defined, then s'.known_aborts(π) = s.known_aborts(π).*

The next lemma says that the known_aborts set for an event π includes all events that directly affect π, and that the known_aborts set for π includes T if π is directly affected by an ABORT(T) event.

LEMMA 15. *Let β be a finite Argus behavior that can leave the Argus database in state s.*

(1) *If ϕ and π are events in β such that ϕ directly affects π in β, then s.known_aborts(ϕ) $\subseteq$ s.known_aborts(π).*
(2) *If π is directly affected by an ABORT(T) event in β, then $T \in$ s.known_aborts(π).*

PROOF. Immediate by the definition of the Argus database steps and Lemmas 13 and 14. □

The next lemma is the key to the proof of correctness for the Argus system: It says that the known_aborts set for an event π includes all transactions T such that an ABORT(T) event affects π. In other words, the Argus database propagates enough information about aborts so that every event π "knows about" (stores in s.known_aborts(π)) every abort that affects it.

LEMMA 16. *Let β be a finite Argus behavior that can leave the Argus database in state s. If ϕ and π are events in β such that ϕ affects π in β and ϕ is an ABORT(T) event, then $T \in s.known\_aborts(\pi)$.*

PROOF. The proof proceeds by induction on the length of the chain in the directly-affects relation by which ϕ affects π in β. If the length of the chain is 1, then ϕ directly affects π in β. Then, Lemma 15 implies that T $\in$ s.known_aborts(π).

Now suppose that the length of the chain is $k + 1$, where $k \geq 1$. Then there is an event ψ in β such that ϕ affects ψ in β by a chain of length at most k and ψ directly affects π in β. By inductive hypothesis, T $\in$ s.known_aborts(ψ). Lemma 15 implies that s.known_aborts(ψ) $\subseteq$ s.known_aborts(π), so that T $\in$ s.known_aborts(π). □

6.3. SIMULATION OF THE BASIC SYSTEM BY THE ARGUS SYSTEM. The following lemma shows that the information in known_aborts, combined with the precondition on REQUEST_COMMIT actions for accesses, is enough to ensure that the Argus system implements the filtered system.

LEMMA 17. *The Argus system implements the filtered system.*

PROOF. We define a mapping f that assigns to each state s of the Argus system the singleton set f(s) consisting of the state of the filtered system that is the same except for the omission of the s.known_aborts component of the Argus database state. We must show that f is a possibilities mapping. Condition 1 is easy to check. For condition 2, suppose that s′ is a reachable state of the Argus system, t′ $\in$ f(s′) is a reachable state of the filtered system, and (s′, π, s) is a step of the Argus system. The only interesting case to check is when π = REQUEST_COMMIT(T, v), where T is an access to an object X. In this case, we claim that (t′, π, t) is a step of the filtered system, where t is the single element of f(s).

To show that (t′, π, t) is a step of the filtered system, we must show the four conditions defining these steps. The first three are immediate from the definition of the steps of the Argus database. To see the fourth, suppose that T′ is an ancestor of T. We must show that no ABORT(T′) event affects an event of object X in t′.history. Suppose the contrary, that an ABORT(T′) event ϕ affects an event ψ of object X in t′.history. Then, ϕ also affects ψ in s′.history. By Lemma 16, T′ $\in$ s.known_aborts(ψ). But this violates the precondition for π in the Argus database. Therefore, π is enabled in t′. □

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

THEOREM 18. *Let β be an Argus behavior and let T be a transaction name. Then there exists a basic behavior γ such that T is not an orphan in γ and $\gamma|T = \beta|T$.*

PROOF. Immediate by Lemma 17 and Theorem 9. □

As for the filtered system, we obtain an important corollary about serial correctness.

COROLLARY 19. *If the basic system is serially correct for nonorphans, then the Argus system is serially correct.*

Again, we give a version of the preceding corollary in which the dependency of the Argus system on the basic system is made explicit. If B is a basic system and R′ is a family of directly-affects relations for B, then let Argus(B, R′) be the corresponding Argus system; it is composed of the same transaction automata and the Argus database that is constructed from the given basic database, using the given family of relations R′.

COROLLARY 20. *Suppose B is a basic system and R′ is a family of directly-affects relations for B. If B is serially correct for nonorphans, then Argus(B, R′) is serially correct.*

7. *Strictly Filtered Systems*

The orphan management algorithm described in [17] actually ensures a stronger property than does the Argus algorithm. It ensures that REQUEST_COMMIT can never occur for an orphan access, whereas the Argus algorithm merely ensures that no such REQUEST_COMMIT can occur if the access can "observe" that it is an orphan. In this section, we define the *strictly filtered database*, which allows a REQUEST_COMMIT to occur for an access only if no ancestor has aborted. (Compare this to the filtered database, which allows an access to REQUEST_COMMIT if an ancestor has aborted as long as the access is not affected by the abort.) We then define the strictly filtered system, which is composed of transactions and the strictly filtered controller, and show that the strictly filtered system implements the filtered system. In the next section, we describe formally the algorithm from [17] and show that it implements the strictly filtered system.

7.1. THE STRICTLY FILTERED DATABASE. The strictly filtered database is similar to the filtered database; it has the same actions, and the same states. A triple (s', π, s) is a step of the strictly filtered database if and only if the following conditions hold.

(1) (s′.basic_state, π, s.basic_state) is a step of the basic database,
(2) s.history = s′.historyπ if π is a basic action,
(3) s.history = s′.history if π is not a basic action,
(4) If π = REQUEST_COMMIT(T, v) where T is an access, and T′ is an ancestor of T, then no ABORT(T′) event occurs in s′.history.

Thus, at the point where the REQUEST_COMMIT of an access is about to occur, an explicit test is performed to verify that there is no preceding abort of any ancestor of the access.

7.2. THE STRICTLY FILTERED SYSTEM. The *strictly filtered system* is the composition of transactions and the strictly filtered database. Executions, schedules, and behaviors of the strictly filtered system are *strictly filtered executions*, *schedules*, and *behaviors*, respectively.

LEMMA 21. *The strictly filtered system implements the basic system.*

PROOF. The proof is similar to that of Lemma 7. □

7.3. SIMULATION OF THE BASIC SYSTEM BY THE STRICTLY FILTERED SYSTEM

LEMMA 22. *The strictly filtered system implements the filtered system.*

PROOF. We define a mapping f that assigns to each state s of the strictly filtered system the singleton set f(s) that consists of the same state. We must show that f is a possibilities mapping. Condition 1 is easy to check. For condition 2, suppose that s′ is a reachable state of the strictly filtered system, $t' \in f(s')$ is a reachable state of the filtered system, and (s', π, s) is a step of the strictly filtered system. As before, the only interesting case to check is when π = REQUEST_COMMIT(T, v), where T is an access to an object X. In this case, we claim that (t', π, t) is a step of the filtered system, where t is the unique element of f(s).

To show that (t', π, t) is a step of the filtered system, we must show the four conditions defining these steps. The first three are immediate from the definition of the steps of the strictly filtered database. To see the fourth, suppose that T′ is an ancestor of T. We must show that no ABORT(T′) event affects an event of object X in t′.history. But t′.history = s′.history, and by the preconditions for π in the strictly filtered database, no ABORT(T′) event occurs in s′.history. Therefore, no ABORT(T′) event affects an event of object X in t′.history. Thus, π is enabled in t′. □

Strictly filtered systems, like filtered systems and Argus systems, prevent orphans from discovering that they are orphans.

THEOREM 23. *Let β be a strictly filtered behavior and let T be a transaction name. Then there exists a basic behavior γ such that T is not an orphan in γ and $\gamma|T = \beta|T$.*

PROOF. Immediate by Lemma 22 and Theorem 9. □

COROLLARY 24. *If the basic system is serially correct for nonorphans, then the strictly filtered system is serially correct.*

If B is a basic system, let Strictly-Filtered(B) be the corresponding strictly filtered system; it is composed of the same transaction automata and the strictly filtered database that corresponds to the given basic database.

COROLLARY 25. *Suppose that B is a basic system. If B is serially correct for nonorphans then Strictly-Filtered(B) is serially correct.*

8. *Clocked Systems*

In this section, we describe formally the orphan management algorithm from [17]. We do this by defining the *clocked database*, which uses a global clock to ensure that transactions do not abort until all their descendant accesses have stopped running. We then define the clocked system, which is composed of transactions and the clocked database. Finally, we show that the clocked system implements the strictly filtered system, and thus simulates the basic system in the same way as the previously mentioned systems do.

8.1. THE CLOCKED DATABASE. The clocked database maintains a quiesce time for each access transaction and a release time for every transaction. An access transaction is allowed to REQUEST_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 request to commit.

If quiesce and release times are fixed in advance, some transactions may be forced to abort unnecessarily as their quiesce times expire, and aborts may need to be delayed until release times are reached. It is possible to obtain extra flexibility by providing actions in the clocked database for adjusting quiesce and release times.

The action signature of the clocked database is the same as that of the basic database, except that the clocked database has three additional kinds of internal actions. The new actions are:

Internal Actions:
TICK
ADJUST_QUIESCE(T), T an access
ADJUST_RELEASE(T), T any transaction

The TICK action advances the clock, while the two ADJUST actions 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 clocked database consists of components *basic_state*, *aborted*, *clock*, *quiesce*, and *release*. Here, basic_state is a state of the basic database, initialized at an initial state of the basic database. The component *aborted* is a set of transactions, initially empty. The component *clock* is a real number, initialized arbitrarily. The component *quiesce* is a total mapping from access transaction names to real numbers, and the component *release* is a total mapping from all transaction names to real numbers. The initial values of quiesce and release are arbitrary, subject to the following condition: for all transaction names T and T′, where T is an access and T′ is an ancestor of T, quiesce(T) $\leq$ release(T′).

A triple (s′, π, s) is a step of the clocked database if and only if the following conditions hold:

(1) If π is an action of the basic system, then (s′.basic_state, π, s.basic_state) is a step of the basic database,
(2) If π is a TICK, ADJUST_QUIESCE, or ADJUST_RELEASE action, then s.basic_state = s′.basic_state,
(3) If π = ABORT(T), then
  (a) s.aborted = s′.aborted $\cup$ {T},
  (b) s′.release(T) $\leq$ s′.clock,
(4) If π is not an abort action, then s.aborted = s′.aborted,
(5) If π = REQUEST_COMMIT(T, v) where T is an access, then s′.clock < s′.quiesce(T),
(6) If π = TICK, then s′.clock < s.clock,
(7) If π is not a TICK action, then s.clock = s′.clock,
(8) If π = ADJUST_RELEASE(T), then
  (a) if T $\in$ s′.aborted, then s.release(T) $\leq$ s′.clock,
  (b) s′.quiesce(T′) $\leq$ s.release(T) for all T′ $\in$ descendants(T) $\cap$ accesses,
  (c) s.release(T′) = s′.release(T′) for all T′ $\neq$ T,

(9) If π is not an ADJUST_RELEASE action, then s.release = s′.release,
(10) If π = ADJUST_QUIESCE(T), then
  (a) s.quiesce(T) $\leq$ s′.release(T′) for all T′ $\in$ ancestors(T),
  (b) s.quiesce(T′) = s′.quiesce(T′) for all T′ $\neq$ T,
(11) If π is not an ADJUST_QUIESCE action, then s.quiesce = s′.quiesce.

LEMMA 26. *Let β be a finite schedule of the clocked database that can leave the clocked database in state s.*

(1) *If $T \in s.aborted$, then $s.release(T) \leq s.clock$,*
(2) *For all accesses T and all ancestors T' of T, $s.quiesce(T) \leq s.release(T')$.*

PROOF. Straightforward by induction. □

8.2. THE CLOCKED SYSTEM. The *clocked system* is the composition of transactions and the clocked database. External actions of the clocked system are called *clocked actions*. Executions, schedules, and behaviors of a clocked system are called *clocked executions*, *schedules*, and *behaviors*, respectively.

LEMMA 27. *The clocked system implements the basic system.*

PROOF. The proof is similar to that of Lemma 7. □

8.3. SIMULATION OF THE BASIC SYSTEM BY THE CLOCKED SYSTEM

LEMMA 28. *The clocked system implements the strictly filtered system.*

PROOF. We define a mapping f that assigns to each state s of the clocked system the set f(s) of states t of the strictly filtered system such that t.basic_state = s.basic_state and t.history is a sequence of basic actions in which the set of transaction names T for which ABORT(T) occurs in t.history is exactly s.aborted. We must show that f is a possibilities mapping. Condition 1 is easy to check. For condition 2, suppose that s′ is a reachable state of the clocked system, t′ $\in$ f(s′) is a reachable state of the strictly filtered system, and (s′, π, s) is a step of the clocked system.

There are two interesting cases to check: where π = ABORT(T) and where π = REQUEST_COMMIT(T, v) for an access T. In either case, we claim that (t′, π, t) is a step of the strictly filtered system, where t is the state of the strictly filtered system in which t.basic_state = s.basic_state and t.history = t′.historyπ, and we also claim that t $\in$ f(s).

If π = ABORT(T), it is easy to see that (t′, π, t) is a step of the strictly filtered system. To show t $\in$ f(s), note that since t′ $\in$ f(s′), the set of transaction names U for which ABORT(U) occurs in t′.history is exactly s′.aborted. Then, the set of transaction names with aborts in t.history is exactly s′.aborted $\cup$ {T}, which is equal to s.aborted. Thus, t $\in$ f(s).

If π = REQUEST_COMMIT(T, v), where T is an access, then it is easy to see the first three conditions of the definition of strictly filtered database steps. For the fourth condition, we must show that if T′ is an ancestor of T, then no ABORT(T′) event occurs in t′.history. So suppose the contrary, that T′ is an ancestor of T and ABORT(T′) occurs in t′.history. Since t′ $\in$ f(s′), we have T′ $\in$ s′.aborted. Since π is enabled in s′, s′.clock < s′.quiesce(T). Lemma 26 implies that s′.release(T′) $\leq$ s′.clock and also that s′.quiesce(T) $\leq$ s′.release(T′). Thus, s′.quiesce(T) $\leq$ s′.clock, a contradiction. It follows that (t′, π, t) is a step of the strictly filtered system.

Since $t' \in f(s')$, the set of transaction names U for which ABORT(U) occurs in t'.history is exactly s'.aborted. Then, the set of transaction names with aborts in t.history is exactly s'.aborted = s.aborted. Thus, $t \in f(s)$. □

THEOREM 29. *Let β be a clocked behavior and let T be a transaction name. Then there exists a basic behavior γ such that T is not an orphan in γ and $\gamma|T = \beta|T$.*

PROOF. By Lemma 28 and Theorem 23. □

COROLLARY 30. *If the basic system is serially correct for nonorphans, then the clocked system is serially correct.*

If B is a basic system, then let Clocked(B) be the corresponding clocked system; it is composed of the same transaction automata and the clocked database of the appropriate type.

COROLLARY 31. *Suppose B is a basic system. If B is serially correct for nonorphans, then Clocked(B) is serially correct.*

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 [17]). The adapted algorithms can be described and analyzed in a manner similar to that used for the Argus algorithm.

9. *Examples*

In this section, we describe two important kinds of basic systems, together with a family of affects relations (and a generating family of directly-affects relations) for each of them. The first kind of system is a *generic system*. It is suitable for modeling locking algorithms and has been studied in [4]. The second is a *pseudotime system*. It is suitable for modeling timestamp algorithms and has been studied in [2].

9.1. GENERIC SYSTEMS.

A generic system consists of a collection of transaction automata, one for each nonaccess transaction name, a collection of *generic object automata*, one for each object name, and a single *generic controller automaton*. The interactions between the components are as follows:

The transaction interface is exactly as before. The generic object automaton for X has CREATE(T) input actions and REQUEST_COMMIT(T, v) output actions for each access T to X and each return value v. It also has INFORM_COMMIT_AT(X)OF(T) and INFORM_ABORT_AT(X)OF(T) input actions for each transaction T; these actions inform the object X of the fates (commit or abort) of completed transactions. The object uses this information in carrying out concurrency control and recovery; for example, an INFORM_ABORT of T might cause the object to release locks held by T.

The external actions of the generic controller are similar to those required of all basic databases: It has the inputs and outputs required of a basic database, except that REQUEST_COMMIT actions for access transactions are inputs to the generic controller; in addition, it has INFORM_COMMIT and INFORM_ABORT actions as outputs.

We model the generic controller as a specific automaton, particular to the system type. The object automata, however, like the transaction automata, are only partially specified. Their signature is as described above, and they are

constrained to preserve an appropriately defined well-formedness property. Otherwise, they are unconstrained. In particular, the semantics of their operations is immaterial to our discussion. In this paper, we are concerned only with whether the entire generic database is serially correct for nonorphans, in which case the various algorithms presented earlier guarantee the transformed system is serially correct for all transaction names. The problem of ensuring that specific generic systems are serially correct for nonorphans is addressed elsewhere [4].

9.1.1. *Generic Actions and Well-Formedness.* For a generic system, we extend the object mapping as follows. Define object(π) = X if π is an INFORM_COMMIT_AT(X)OF(T) or INFORM_ABORT_AT(X)OF(T) action. We define the *generic actions* to be the serial actions, plus the INFORM_COMMIT and INFORM_ABORT actions.

Now we define *generic object well-formedness*. Let X be any object name. A sequence β of generic actions π with object(π) = X is defined to be *generic object well-formed* for X provided that the following conditions hold.

(1) There is at most one CREATE(T) event in β for any transaction T.
(2) There is at most one REQUEST_COMMIT event in β for any transaction T.
(3) If there is a REQUEST_COMMIT event for access transaction T in β, then there is a preceding CREATE(T) event in β.
(4) There is no transaction T for which both an INFORM_COMMIT_AT(X)OF(T) event and an INFORM_ABORT_AT(X)OF(T) event occur.
(5) If an INFORM_COMMIT_AT(X)OF(T) event occurs in β and T is an access to X, then there is a preceding REQUEST_COMMIT event for T.

The following simple lemma shows a connection between transaction well-formedness for accesses and generic object well-formedness.

LEMMA 32. *Suppose β is a sequence of generic actions π with object(π) = X. If β is generic object well-formed for X and T is an access to X, then $\beta|T$ is transaction well-formed for T.*

Notice that generic object well-formedness allows multiple (repeated) INFORM events for a transaction. Also, generic object well-formedness does not constrain INFORM events to follow the corresponding completion events (COMMIT or ABORT); this constraint involves the entire system, not just individual objects, and is captured by the generic controller as described below.

9.1.2. *Generic Object Automata.* A *generic object automaton* G for an object name X is an I/O automaton with the following external action signature.

Input:
CREATE(T), for T an access to X
INFORM_COMMIT_AT(X)OF(T), for T any transaction name
INFORM_ABORT_AT(X)OF(T), for T any transaction name
Output:
REQUEST_COMMIT(T, v), for T an access to X and v a value

In addition, G may have an arbitrary set of internal actions. G is required to preserve generic object well-formedness. Except for this well-formedness requirement, generic object automata can be chosen arbitrarily.

Generic objects are similar to the abstract objects (instances of abstract data types) of Argus and other object-oriented systems. A generic object provides a set of accesses through which other transactions can observe and change the object's state. (These accesses can be thought of as instances of the operations usually assumed for objects in object-oriented systems.) Accesses can be invoked by concurrent transactions, and transactions can abort; thus, in generic transaction-processing systems that guarantee serial correctness, generic objects must provide synchronization and recovery. The objects studied in [12] and [13], which use an exclusive locking variation of Moss's algorithm [18] for synchronization combined with version stacks for recovery, are examples of generic objects that provide synchronization and recovery sufficient to ensure serial correctness for nonorphans. Similarly, the more general objects studied in [4], which use a commutativity-based locking algorithm that permits concurrent updates, are also generic objects that ensure serial correctness for nonorphans.

9.1.3. *Generic Controller.* The third kind of component in a generic system is the generic controller. The generic controller is also modeled as an I/O automaton. The transactions and generic objects have been specified to be any automata whose actions and behavior satisfy certain 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 passes requests for the creation of subtransactions to the appropriate recipients, makes decisions about the commit or abort of transactions, passes reports about the completion of children back to their parents, and informs objects of the fate of transactions. It allows concurrency and aborts, and leaves the task of coping with them to the generic objects.

The generic controller is a very nondeterministic automaton. It may delay passing requests or reports or making decisions for arbitrary lengths of time, and may decide at any time to abort a transaction whose creation has been requested (but that has not yet completed). The generic controller can be implemented in many different ways by controllers that make specific choices from among the many nondeterministic possibilities. For instance, Moss [18] describes a distributed implementation of the generic controller that copes with node and communication failures yet still commits a subtransaction whenever possible. Our results apply a fortiori to all implementations of the generic controller obtained by restricting the nondeterminism.

The generic controller has the following action signature.

Input:
REQUEST_CREATE(T), $T \neq T_0$
REQUEST_COMMIT(T, v)

Output:
CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T, v), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$
INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
INFORM_ABORT_AT(X)OF(T), $T \neq T_0$

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

Each state s of the generic controller consists of six sets: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted, and s.reported. The set s.commit_requested is a set of (transaction, value) pairs, and the others are sets of transaction names. All are empty in the start state except for create_requested, which is $\{T_0\}$. Define s.completed = s.committed $\cup$ s.aborted.

The transition relation of the generic controller consists of exactly those triples (s', π, s) satisfying the preconditions and yielding the effects described below, where π is the indicated action. We include in the effects only those conditions on the state s that may change with the action. If a component of s is not mentioned in the effects, it is implicit that the set is the same in s′ and s.

REQUEST_CREATE(T), $T \neq T_0$
Effect:
  s.create_requested = s′.create_requested $\cup$ {T}

REQUEST_COMMIT(T, v)
Effect:
  s.commit_requested = s′.commit_requested $\cup$ {(T, v)}

CREATE(T)
Precondition:
  $T \in$ s′.create_requested − s′.created
Effect:
  s.created = s′.created $\cup$ {T}

COMMIT(T), $T \neq T_0$
Precondition:
  (T, v) $\in$ s′.commit_requested for some v
  $T \notin$ s′.completed
Effect:
  s.committed = s′.committed $\cup$ {T}

ABORT(T), $T \neq T_0$
Precondition:
  $T \in$ s′.create_requested − s′.completed
Effect:
  s.aborted = s′.aborted $\cup$ {T}

REPORT_COMMIT(T, v), $T \neq T_0$
Precondition:
  $T \in$ s′.committed
  (T, v) $\in$ s′.commit_requested
  $T \notin$ s′.reported
Effect:
  s.reported = s′.reported $\cup$ {T}

REPORT_ABORT(T), $T \neq T_0$
Precondition:
  $T \in$ s′.aborted
  $T \notin$ s′.reported

Effect:
s.reported = s′.reported ∪ {T}

INFORM_COMMIT_AT(X)OF(T), T ≠ T_0
Precondition:
T ∈ s′.committed

INFORM_ABORT_AT(X)OF(T), T ≠ T_0
Precondition:
T ∈ s′.aborted

The generic controller assumes that its input actions, 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 action. The precondition of the CREATE action indicates that a given transaction will be created at most once; the effect of the CREATE is to record that the creation has occurred. Similarly, the effect of a COMMIT or ABORT action is to record that the action has occurred. REPORT_COMMIT, REPORT_ABORT, INFORM_COMMIT, and INFORM_ABORT actions can be generated at any time after the corresponding COMMIT and ABORT actions have occurred. The precondition for a COMMIT action ensures that a transaction only commits if it has requested to do so, and it has not already completed (committed or aborted). The precondition for an ABORT action ensures that a transaction will be aborted only if a REQUEST_CREATE has occurred for it and it has not already completed. 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 following lemma follows easily by induction, using simple invariants maintained by the generic controller. (See [4].)

LEMMA 33

(1) *Let T be any transaction name. Then the generic controller preserves transaction well-formedness for T.*
(2) *Let X be any object name. Then the generic controller preserves generic object well-formedness for X.*
(3) *The generic controller preserves basic database well-formedness.*

9.1.4. *Generic Database.* A *generic database* is the composition of a strongly compatible set of automata indexed by the union of the set of object names and the singleton set {GC} (for "generic controller"). Associated with each object name X is a generic object automaton G_X for X, and associated with the name GC is the generic controller automaton for the system type.

LEMMA 34. *If β is a behavior of a generic database, then for every object name X, $\beta|X$ is generic object well-formed.*

PROOF. Let X be an object name. Of the components of the generic database, only G_X and the generic controller have external actions π for which object(π) = X. The other components thus trivially preserve generic object well-formedness for X. By explicit assumption, G_X preserves generic object well-formedness for X, and by Lemma 33, the generic controller

preserves generic object well-formedness for X. The result follows from Proposition 4. □

LEMMA 35. *Let B be a generic database. Then B preserves basic database well-formedness.*

PROOF. A simple case analysis. The only subtle case is when $\beta\pi$ is a behavior of B, with β basic database well-formed and π a REQUEST_COMMIT(T, v) event for an access T to X. The argument that $\beta\pi|T$ is transaction well-formed for T depends upon the fact that $\beta\pi|X$ is generic object well-formed for X, as shown in Lemma 34. □

It follows that the generic database is an example of a basic database.

9.1.5. *Generic Systems.* A *generic system* is the composition of a strongly compatible set of automata indexed by the union of the set of nonaccess transaction names, the set of object names, and the singleton set {GC}. Associated with each nonaccess transaction name T is a transaction automaton A_T for T. Associated with each object name X is a generic object automaton G_X for X. Finally, associated with the name GC is the generic controller automaton for the system type.

When the particular generic system is clear from context, we call its executions, schedules, and behaviors the *generic executions*, *generic schedules*, and *generic behaviors*, respectively.

9.1.6. *A Family of Affects Relations.* Now we define a family $R = \{R_\beta\}$ of affects relations for any particular generic system B. We do this by first defining a family $R' = \{R'_\beta\}$ of directly-affects relations for B, and then taking transitive closures. For a sequence β of generic actions, define the relation R'_β to be the relation containing the pairs (ϕ, π) of events such that ϕ occurs before π in β, and at least one of the following holds:

—transaction(ϕ) = transaction(π) and π is an output event of the transaction,
—object(ϕ) = object(π) and π is a REQUEST_COMMIT(T, v) event,
—ϕ is a REQUEST_CREATE(T) and π a CREATE(T) event,
—ϕ is a REQUEST_COMMIT(T, v) and π a COMMIT(T) event,
—ϕ is a REQUEST_CREATE(T) and π an ABORT(T) event,
—ϕ is a COMMIT(T) and π a REPORT_COMMIT(T, v) event,
—ϕ is an ABORT(T) and π a REPORT_ABORT(T) event,
—ϕ is a COMMIT(T) and π an INFORM_COMMIT_AT(X)OF(T) event,
—ϕ is an ABORT(T) and π an INFORM_ABORT_AT(X)OF(T) event.

Now define the relation R_β to be the transitive closure of R'_β. It is easy to see that R_β is an irreflexive partial order.

The idea is that ϕ directly affects π if they both occur at the same transaction or object (and π is an output of the transaction, or a REQUEST_COMMIT of the object), or if they involve different transactions or objects but the generic system requires ϕ to occur before π can occur. This notion of one event 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 events involve different transactions or objects, the preconditions for π in the generic controller require ϕ to occur if ϕ directly affects π. If the events occur at the same transaction or object, however, it might be that ϕ happens to occur

before π, yet that the particular transaction or object does not require ϕ to occur before π. In the absence of more information about the particular transactions or objects used in a system, however, it is difficult to say more about the ways in which one event can affect another. Thus, we make the "safe" choice of assuming an effect whenever one could occur. Fortunately, the orphan management algorithms described earlier in this paper are essentially independent of the particular transactions and objects used in a system, and do not rely on more information about them.

The next few lemmas show that the family $\{R_\beta\}$ defined above is a family of affects relations for B.

LEMMA 36. *If β is a finite behavior of generic system B and γ is an R_β-closed subsequence of β, then γ is a behavior of B.*

PROOF. For each nonaccess transaction name T, let A_T be the transaction automaton for T in B, and for each object name X, let G_X be the generic object automaton for X in B. By Proposition 2, it suffices to show that $\gamma|T$ is a behavior of A_T for all nonaccess transaction names T, that $\gamma|X$ is a behavior of G_X for all object names X, and that γ is a behavior of the generic controller. We show these in turn.

First, suppose that T is a nonaccess transaction name. If $\gamma|T$ contains no output events of A_T, then the input-enabling property implies that $\gamma|T$ is a behavior of A_T. So assume that there is at least one output event of A_T in $\gamma|T$, and let π be the last such event. Let γ' be the prefix of γ ending with π. Since γ is closed in β, it follows from the definition of R_β that γ contains all events of T that precede π in β. Thus, $\gamma'|T$ is a prefix of $\beta|T$ and so is a behavior of A_T. Since $\gamma|T$ differs from $\gamma'|T$ only by the possible inclusion of some final input events of A_T, the input-enabling property implies that $\gamma|T$ is a behavior of A_T.

A similar argument shows that if X is an object name, then $\gamma|X$ is a behavior of G_X.

Now we show that γ is a behavior of the generic controller. Note that the generic controller is deterministic in the sense that for a given state s′ and action π, there is at most one state s such that (s', π, s) is a step of the generic controller. We proceed by induction on the lengths of prefixes δ of γ. The basis, where the length of δ is 0, is obvious. So suppose that $\delta = \delta'\pi$, where π is a single event. Let $\beta'\pi$ be the prefix of β ending with π. Let s′ be the state of the generic controller after δ'. We consider cases, showing in each case that π is enabled in s′.

(1) π = CREATE(T)

Then, Lemma 35 and Proposition 5 imply that REQUEST_CREATE(T) occurs in β', and no CREATE(T) occurs in β'. Since γ is R_β-closed in β, REQUEST_CREATE(T) also occurs in δ'. Similarly, no CREATE(T) occurs in δ'. It follows that π is enabled in s′.

(2) π = COMMIT(T)

Then, Lemma 35 and Proposition 5 imply that β' contains REQUEST_COMMIT(T, v) and contains no completion events for T. Since γ is R_β-closed in β, δ' contains REQUEST_COMMIT(T, v), and does not contain a completion event for T. It follows that π is enabled in s′.

(3) π = ABORT(T)

Then β' contains REQUEST_CREATE(T) and contains no completion events for T, so δ' contains REQUEST_CREATE(T) and no completion events for T. Thus, π is enabled in s'.

(4) π = REPORT_COMMIT(T, v)

Then β' contains COMMIT(T) and REQUEST_COMMIT(T, v) and contains no report events for T. Therefore, the same is true of δ', so π is enabled in s'.

(5) π = REPORT_ABORT(T)

Similar to the preceding arguments.

(6) π = INFORM_COMMIT_AT(X)OF(T)

Similar to the preceding arguments.

(7) π = INFORM_ABORT_AT(X)OF(T)

Similar to the preceding arguments. □

LEMMA 37. *The family* $\{R_\beta\}$ *is a family of affects relations for B.*

PROOF. The first two properties of families of affects relations are immediate from the definition of $\{R_\beta\}$. The third property is proved in the lemma above.

The fourth property requires that whenever an ABORT(T) event ϕ affects an event π of certain types, there is a specific type of intervening event ψ that is also affected by ϕ. To see that this property is satisfied, note that R_β is defined as the transitive closure of the directly-affects relation R'_β. Each type of event π of interest is only directly affected by earlier events that satisfy the restrictions on ψ. Thus, if π is affected by ϕ, it must be so affected by the transitive closure over a chain of directly-affects relations in which an event ψ of the appropriate type occurs. □

9.1.7. *Applying Orphan Management Algorithms to Generic Systems.* Since we have shown that generic systems are instances of basic systems, and that the directly-affects relation we defined above generates a family of affects relations, we may apply our general results for orphan management algorithms to generic systems. Before we state these results, we illustrate one orphan management transformation by describing explicitly the steps of the system obtained by applying the Argus algorithm to a generic system.

If B is a generic system and R'_β is the family of directly-affects relations given above, then the steps of Argus(B, R'_β) are defined as follows:

—(s'.basic_state, π, s.basic_state) is a step of the generic database,
—s.history = s'.historyπ if π is a basic action,
—s.history = s'.history if π is not a basic action,
—If $\phi \neq \pi$, then s.known_aborts(ϕ) = s'.known_aborts(ϕ),
—If π is a REQUEST_CREATE(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all ϕ in s'.history such that transaction(ϕ) = parent(T),
—If π is a REQUEST_COMMIT(T, v) action, where T is a nonaccess

transaction name, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all ϕ in s'.history such that transaction(ϕ) = T,
—If π is a REQUEST_COMMIT(T, v) action, where T is an access transaction name, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all ϕ in s'.history such that object(ϕ) = object(T),
—If π is a CREATE(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all REQUEST_CREATE(T) events ϕ in s'.history,
—If π is a COMMIT(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all REQUEST_COMMIT(T, v) events ϕ in s'.history,
—If π is an ABORT(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all REQUEST_CREATE(T) events ϕ in s'.history,
—If π is a REPORT_COMMIT(T, v) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all COMMIT(T) events ϕ in s'.history,
—If π is a REPORT_ABORT(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all ABORT(T) events ϕ in s'.history, and T $\in$ s.known_aborts(π),
—If π is an INFORM_COMMIT_AT(X)OF(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all COMMIT(T) events ϕ in s'.history,
—If π is an INFORM_ABORT_AT(X)OF(T) action, then s'.known_aborts(ϕ) $\subseteq$ s.known_aborts(π) for all ABORT(T) events ϕ in s'.history, and T $\in$ s.known_aborts(π),
—If π is a REQUEST_COMMIT(T, v) action for an access T to object X, then there is no ancestor of T in s'.known_aborts(ϕ), for any event ϕ of object X in s'.history.

The filtered database uses global information about the history to prevent the REQUEST_COMMIT of an access from occurring if it would be affected by an ABORT of an ancestor. The Argus database uses more local information, which is obtained by propagating the known_aborts sets from each event to any later events that it directly affects. For example, consider the application above of the Argus algorithm to a generic database. The known_aborts set for a REQUEST_CREATE(T) action is obtained from the known_aborts sets for all preceding events at parent(T). Since REQUEST_CREATE(T) is generated by parent(T), the known_aborts set for it can easily be computed with information available locally at parent(T) when the REQUEST_CREATE(T) action occurs. A similar situation arises with REQUEST_COMMIT actions, which are outputs of transactions and objects. The other actions, which are outputs of the generic controller, are directly affected by exactly one preceding event. Thus, the known_aborts set for one of these actions can easily be computed from the known_aborts set for the single event that precedes it. For instance, the known_aborts set for a CREATE(T) event can be obtained directly from the known_aborts set for the preceding REQUEST_CREATE(T) event. If the two events occur at the same site in a network, this information would be available locally; if they occur at different sites, it could be sent in the message used to transmit the REQUEST_CREATE event to the site that performs the CREATE event.

The following corollary shows that the Argus algorithm and the clocked algorithm from [17] can both be used for a generic system:

COROLLARY 38. *Let B be a generic system, and R and R' the family of affects*

and directly-affects relations defined above. If B is serially correct for nonorphans, then the following are true:

—*Filtered*(B, R) *is serially correct*,
—*Argus*(B, R') *is serially correct*,
—*Strictly-Filtered*(B) *is serially correct*,
—*Clocked*(B) *is serially correct*.

9.2. PSEUDOTIME SYSTEMS. The essential feature of systems using timestamps is the explicit construction of a sibling order representing the intended serialization of an execution. This order is represented in terms of intervals of *pseudotime*, an arbitrarily chosen totally ordered set. Formally, we let P be the set of pseudotimes, ordered by $<$. We represent pseudotime intervals as half-open intervals $[p, q)$ in P, and refer to them using capital letters. If $P = [p, q)$, then we write P_{min} for p and P_{max} for q. If P and Q are intervals of pseudotime, we write $P < Q$ if $P_{max} \leq Q_{min}$. Clearly, if $P < Q$, then P and Q are disjoint.

A pseudotime system consists of a collection of transaction automata, one for each nonaccess transaction name, a collection of pseudotime object automata, one for each object name, and a single pseudotime controller automaton. The interactions between the components are as follows. The transaction interface is exactly as before. A pseudotime object automaton for X has the same actions as a generic object automaton, with the addition of INFORM_TIME_AT(X)OF(T, p) input actions to inform the object that pseudotime p has been assigned to an access transaction T. The pseudotime controller has the same actions as the generic controller, with the addition of INFORM_TIME_AT(X)OF(T, p) output actions (for access transactions T) and ASSIGN_PSEUDOTIME(T, P) output actions (for all transactions T) by which the controller assigns the pseudotime range P to transaction T.

9.2.1. *Pseudotime Actions and Well-Formedness.* The object mapping for a pseudotime system is the same as that for a generic system, with the addition that we define object(π) = X if π is an INFORM_TIME_AT(X)OF(T, p) action. We define the *pseudotime actions* to be the generic actions, plus the INFORM_TIME and ASSIGN_PSEUDOTIME actions.

Now we define "pseudotime object well-formedness." Let X be any object name. A sequence of pseudotime actions π with object(π) = X is defined to be *pseudotime object well-formed* for X provided that the following conditions hold:

(1) There is at most one CREATE(T) event in β for any transaction T,
(2) There is at most one REQUEST_COMMIT event in β for any transaction T,
(3) If there is a REQUEST_COMMIT event for T in β, then there is a preceding CREATE(T) event and also a preceding INFORM_TIME_AT(X)OF(T, p) in β,
(4) There is no transaction T for which there are two different pseudotimes, p and p′, such that INFORM_TIME_AT(X)OF(T, p) and INFORM_TIME_AT(X)OF(T, p′) both occur in β,
(5) There is no pseudotime p for which there are two different transactions, T and T′, such that INFORM_TIME_AT(X)OF(T, p) and INFORM_TIME_AT(X)OF(T′, p) both occur in β,

(6) There is no transaction T for which both an INFORM_COMMIT_AT(X)OF(T) event and an INFORM_ABORT_AT(X)OF(T) event occur,
(7) If an INFORM_COMMIT_AT(X)OF(T) event occurs in β and T is an access to X, then there is a preceding REQUEST_COMMIT event for T.

9.2.2. *Pseudotime Object Automata.* A *pseudotime object automaton* P for an object name X is an I/O automaton with the following external action signature.

Input:
CREATE(T), T an access to X
INFORM_COMMIT_AT(X)OF(T)
INFORM_ABORT_AT(X)OF(T)
INFORM_TIME_AT(X)OF(T, p), T an access to X, $p \in P$

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

In addition, P may have an arbitrary set of internal actions. P is required to preserve pseudotime object well-formedness.

9.2.3. *Pseudotime Controller.* The *pseudotime controller* guarantees that siblings are assigned disjoint intervals of pseudotime, and that each transaction's interval is a subset of that of its parent. The pseudotime controller has the actions of the generic controller together with an extra class of output actions ASSIGN_PSEUDOTIME(T, P) for $T \neq T_0$ and P a pseudotime interval. The purpose of the ASSIGN_PSEUDOTIME actions is to construct, at run-time, a sibling order that specifies the apparent serial ordering of transactions. Also, there is an extra class of actions INFORM_TIME_AT(X)OF(T, p) for access transactions T. A state s of the pseudotime controller has the same components as a state of the generic controller together with an additional component s.interval, which is a partial function from T to the set of pseudotime intervals. In the initial state s_0 of the pseudotime controller s_0.interval = $\{(T_0, P_0)\}$ for some pseudotime interval P_0, and all other components are as in the initial state of the generic controller.

The transaction relation for generic actions is the same as that for the generic controller, except that the actions CREATE(T) and ABORT(T) have an additional precondition: $T \in$ domain(s'.interval). The additional actions are determined as follows:

ASSIGN_PSEUDOTIME(T, P)
Precondition:
$T \in s'$.create-requested
$T \notin$ domain(s'.interval)
$P \subseteq s'$.interval(parent(T))
$P >$ s'.interval(T') for every T' in siblings(T) $\cap$ domain(s'.interval)
Effect:
s.interval = s'.interval $\cup$ {(T, P)}

INFORM_TIME_AT(X)OF(T, p), T an access to X
Precondition:
$(T, P) \in s'$.interval
$p = P_{min}$

The following lemma is straightforward.

LEMMA 39

(1) *Let T be any transaction name. Then the pseudotime controller preserves transaction well-formedness for T,*
(2) *Let X be any object name. Then the pseudotime controller preserves pseudotime object well-formedness for X,*
(3) *The pseudotime controller preserves basic database well-formedness.*

9.2.4. *Pseudotime Database.* A *pseudotime database* is the composition of a strongly compatible set of automata indexed by the union of the set of object names and the singleton set {PC} (for "pseudotime controller"). Associated with each object name X is a pseudotime object automaton P_X for X. Finally, associated with the name PC is the pseudotime controller automaton for the system type.

LEMMA 40. *If β is a behavior of a pseudotime database, then for every object name X, $\beta|X$ is pseudotime object well-formed.*

LEMMA 41. *Let B be a pseudotime database. Then B preserves basic database well-formedness.*

It follows that the pseudotime database is an example of a basic database.

9.2.5. *Pseudotime Systems.* A *pseudotime system* is the composition of a strongly compatible set of automata indexed by the union of the set of nonaccess transaction names, the set of object names, and the singleton set {PC}. Associated with each nonaccess transaction name T is a transaction automaton A_T for T. Associated with each object name X is a pseudotime object automaton P_X for X. Finally, associated with the name PC is the pseudotime controller automaton for the system type.

When the particular pseudotime system is clear from context, we call its exec utions, schedules, and behaviors the *pseudotime executions*, *pseudotime schedules*, and *pseudotime behaviors*, respectively.

9.2.6. *A Family of Affects Relations.* Now we define a family $R = \{R_\beta\}$ of affects relations for any particular pseudotime system B. We do this by first defining a family $R' = \{R'_\beta\}$ of directly-affects relations for B, and then taking transitive closures. For a sequence β of pseudotime actions, define the relation R'_β to be the relation containing the pairs (ϕ, π) of events such that ϕ occurs before π in β, and at least one of the following holds:

—transaction(ϕ) = transaction(π) and π is an output event of the transaction,
—object(ϕ) = object(π) and π is a REQUEST_COMMIT(T, v) event,
—ϕ is a REQUEST_CREATE(T) and π an ASSIGN_PSEUDOTIME(T, P) event,
—ϕ is an ASSIGN_PSEUDOTIME(T, P) and π a CREATE(T) event,
—ϕ is a REQUEST_COMMIT(T, v) and π a COMMIT(T) event,
—ϕ is a REQUEST_CREATE(T) and π an ABORT(T) event,
—ϕ is an ASSIGN_PSEUDOTIME(T, P) and π an ABORT(T) event,
—ϕ is a COMMIT(T) and π a REPORT_COMMIT(T, v) event,
—ϕ is an ABORT(T) and π a REPORT_ABORT(T) event,

—ϕ is a COMMIT(T) and π an INFORM_COMMIT_AT(X)OF(T) event,
—ϕ is an ABORT(T) and π an INFORM_ABORT_AT(X)OF(T) event, or
—ϕ is an ASSIGN_PSEUDOTIME(T, P) event and π an INFORM_TIME_AT(X)OF(T, p) event.

Once again, define the relation R_β to be the transitive closure of R'_β. It is easy to see that R_β is an irreflexive partial order. We claim that the family $\{R_\beta\}$ defined above is a family of affects relations for G.

LEMMA 42. *If β is a behavior of pseudotime system B and γ is an R_β-closed subsequence of β, then γ is a behavior of B.*

PROOF. Analogous to the proof of Lemma 36. □

LEMMA 43. *The family $\{R_\beta\}$ is a family of affects relations for B.*

9.2.7. *Applying Orphan Management Algorithms to Pseudotime Systems.* The following easy corollary shows that the Argus algorithm and the clocked algorithm from [17] can both be used for a pseudotime system:

COROLLARY 44. *Let B be a pseudotime system, and R and R' the family of affects and directly-affects relations defined above. If B is serially correct for nonorphans, then the following are true*:

—*Filtered(B, R) is serially correct,*
—*Argus(B, R') is serially correct,*
—*Strictly-Filtered(B) is serially correct,*
—*Clocked(B) is serially correct.*

10. *Conclusions*

We have defined correctness properties for orphan management algorithms, and have presented precise descriptions and proofs for two algorithms from [10] and [17]. Our proofs are quite simple, and show that the systems exhibit a substantial degree of modularity: the orphan management algorithms can be used in combination with any concurrency control protocol (in basic system form) that is serially correct for nonorphans. The simplicity of our proofs is a direct result of this modularity, and is in sharp contrast to earlier work [6], in which the orphan management algorithm and the concurrency control protocol were not cleanly separated.

Our proofs have an interesting structure. We first define a simple abstract algorithm that uses global information about the history of the system, and show that it ensures that orphans see consistent views. We then formalize the Argus algorithm and the clocked algorithm in a way that requires the use of only local information, and show that each simulates the more abstract algorithm. The simulation proofs are quite simple, and do not require reproving the properties already proved for the abstract algorithm. The correctness of the Argus and clocked algorithms then follows directly from the correctness of the abstract algorithm.

In this paper, we have analyzed only orphans that result from aborts of transactions. Interesting algorithms have also been developed for detecting and eliminating orphans arising from crashes [10, 17]. These algorithms seem more complicated than the algorithms for handling aborts. An open question is whether the known algorithms for handling crash orphans can be analyzed

using techniques similar to those in this paper. In particular, it would be nice to find a similar separation of concerns for those algorithms, so that the crash-orphan algorithms can be understood independently of concurrency control protocols and abort-orphan algorithms. Whether this will be possible is still unknown.

ACKNOWLEDGMENT. 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. Tech. Rep. GIT-ICS-83/23. Georgia Institute of Technology, Atlanta, Ga., Sept. 1983.
2. ASPNES, J., FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of the 14th International Conference on Very Large Data Bases* (Aug.). Morgan-Kaufmann, San Mateo, Calif., 1988, pp. 431–444.
3. BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
4. FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci. 41*, 1 (Aug. 1990), 65–156.
5. GOLDMAN, K., AND LYNCH, N. Quorum consensus in nested transaction systems. In *Proceedings of 6th Annual ACM Symposium on Principles of Distributed Computation* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 27–41. (Expanded version available as Tech. Rep. MIT/LCS/TM-390. Laboratory for Computer Science. Massachusetts Institute of Technology, Cambridge, Mass., May 1987.)
6. GOREE, J. A. Internal consistency of a distributed transaction system with orphan detection. Tech. Rep. MIT/LCS/TR-286. Massachusetts Institute of Technology, Cambridge, Mass., January 1983.
7. HERLIHY, M., LYNCH, N., MERRITT, M., AND WEIHL, W. On the correctness of orphan elimination algorithms. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing* (Pittsburgh, Pa., July). IEEE, New York, 1987, pp. 8–13. (Extended version available as MIT/LCS/TM-329, Laboratory for Computer Science. Massachusetts Institute of Technology, Cambridge, Mass., May 1987.)
8. DETLEFS, D. L., HERLIHY, M. P., AND WING, J. M. Inheritance of synchronization and recovery properties in avalon/C + +. *IEEE Comput. 21,* 12 (Dec. 1988) 57–69.
9. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst. 5,* 3 (July, 1983), 381–404.
10. LISKOV, B., SCHEIFLER, R., WALKER, E. F., AND WEIHL, W. Orphan detection. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing* (July). IEEE, New York, 1987, pp. 2–7.
11. LYNCH, N. A. Concurrency control for resilient nested transactions. *Adv. Comput. Res. 3* (1986), 335–373.
12. LYNCH, N. A., AND MERRITT, M. Introduction to the theory of nested transactions. *Theoret. Comput. Sci. 62* (1988), 123–185.
13. LYNCH, N., AND MERRITT, M. Introduction to the theory of nested transactions. In *Proceedings of the International Conference on Database Theory* (Rome, Italy, Sept.). 1986, pp. 278–305.
14. LYNCH, N., MERRITT, M., WEIHL, W., AND FEKETE, A. *Atomic Transactions*. Morgan-Kaufmann, San Mateo, Calif. To appear, Fall 1992.
15. LYNCH, N., AND TUTTLE, M. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing* (Vancouver, B. C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 137–151. (Expanded version available as Tech. Rep. MIT/LCS/TR-387. Laboratory for Computer Science. Massachusetts Institute of Technology, Cambridge, Mass., April 1987.)
16. LYNCH, N., AND TUTTLE, M. An introduction to input/output automata. *CWI Quarterly 2*, 3 (1989), 219–246.
17. MCKENDRY, M., AND HERLIHY, M. Timestamp-based orphan elimination. *IEEE Trans. Softw. Eng. 15*, 7 (July, 1989).

18. MOSS, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass., 1985.
19. NELSON, B. J. Remote procedure call. Tech. Rep. CMU-CS-81-119. Dept. Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., May 1981.
20. PERL, S. Distributed commit protocols for nested atomic actions. Tech. Rep. MIT/LCS/TR-431. Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1987.
21. PU, C. AND NOE, J. D. Nested transactions for general objects: The Eden implementation. Tech. Rep. TR-85-12-03. Dept. of Computer Science, Univ. of Washington, Seattle, Wash., December 1985.
22. ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, P. M. System level concurrency control for distributed database systems. *ACM Trans. Datab. Syst. 3*, 2 (June 1978), 178–198.
23. SPECTOR, A., AND SWEDLOW, K. *Guide to the Camelot Distributed Transaction Facility: Release 1*. Carnegie-Mellon Univ., Pittsburgh, Pa., Oct. 1987.
24. WALKER, E. F. Orphan detection in the argus system. Tech. Rep. MIT/LCS/TR-326. Massachusetts Institute of Technology, Cambridge, Mass., May 1984.
25. WEIHL, W. E. Specification and implementation of atomic data types. Tech. Rep. MIT/LCS/TR-314. Massachusetts Institute of Technology, Cambridge, Mass., 1984.
26. WEIHL, W. E. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Prog. Lang. Syst. 11*, 2 (Apr. 1989), 249–283.

RECEIVED JUNE 1987; REVISED MARCH 1991; ACCEPTED APRIL 1991