Theoretical Computer Science 62 (1988) 123-185
North-Holland

INTRODUCTION TO THE THEORY OF NESTED TRANSACTIONS

Nancy LYNCH*
Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.

Michael MERRITT
AT&T Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Abstract. A new formal model is presented for studying concurrency and resiliency properties for nested transactions. The model is used to state and prove correctness of a well-known locking algorithm.

1. Introduction

This paper develops the foundation for a general theory of nested transactions. We present a simple formal model for studying concurrency and resiliency in a nested environment. This model has distinct advantages over the many alternatives, the greatest of which is the unification of a subject replete with formalisms, correctness conditions and proof techniques. The authors are presently engaged in an ambitious project to recast the substantial amount of work in nested transactions within this single intuitive framework. These pages contain the preliminary results of that project—a description of the model, and its use in stating and proving correctness conditions for two variations of a well-known algorithm of Moss [22].

The model is based on *I/O automata*, a simple formalization of communicating automata. It is not complex—it is easily presented in a few pages, and easy to understand, given a minimal background in automata theory. Each nested transaction and data object is modelled by a separate I/O automaton. These automata, the system *primitives*, issue requests to and receive replies from some *scheduler*, which is simply another I/O automaton. Simple syntactic constraints on the interactions of these automata ensure, for example, that no transaction requests the creation of the same child more than once. One scheduler, in this case the "serial scheduler", interacts with the transactions and objects in a particularly constrained way. The

* The work of this author was supported by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, and by AT&T Bell Laboratories, Murray Hill, NJ 07974.

0304-3975/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

"serial schedules" of the primitives and the serial scheduler are the basis of our correctness conditions. Specifically, alternative schedulers are required to ensure that nested transaction automata individually have local schedules which they could have in a serial schedule. In essence, each scheduler must "fool" the transactions into believing that the system is executing in conjunction with the serial scheduler.

In the past ten years, an important and substantial body of work has appeared on the design and analysis of algorithms for implementing concurrency control and resiliency in database transaction systems [5, 6, 10, 14, 15, 25, etc.]. Among this has been a number of results dealing with nested transactions [2, 3, 4, 16, 18, 22, 23, 24, etc.]. The present work does not replace these other contributions, but augments them by providing a unifying and mathematically tractable framework for posing and exploring a variety of questions. This previous work uses behavioral specifications of nested transactions, focusing on what nested transactions do, rather than what they are. By answering the question "What is a nested transaction?", I/O automata provide a powerful tool for understanding and reasoning about them.

Some unification is vitally important to further development in this field. The plethora and complexity of existing formalizations is a challenge to the most seasoned researcher. More critically, it belies the argument that nested transactions provide a clean and intuitive tool for organizing distributed databases and more general distributed applications. It is particularly important to provide an intuitive and precise description of nested transactions themselves as, in typical systems, these are the components which the application programmer must implement!

The remainder of this paper is organized as follows. The I/O automaton model is described in Section 2. The rest of the paper contains an extended example, which establishes correctness properties for two related lock-based concurrent schedulers.

Section 3 contains simple definitions for naming nested transactions and objects, and for specifying the operations (interactions) of these components. Simple syntactic restrictions on the orders of these operations are presented, and then a particular system of I/O automata is presented, describing the interactions of nested transactions and objects with a serial scheduler. The interface between the serial scheduler and the transactions provides a basis for the specification of correctness conditions for alternative schedulers. These schedulers would presumably be more efficient than the serial scheduler. The strongest correctness condition, "serial correctness," requires that *all* non-access transactions see serial behavior at their interface with the system. The second condition, "correctness for T_0," only requires that this serial interface be maintained at the interface of the system and the external world. These interfaces also provide simple descriptions of the environment in which nested transactions can be assumed to execute. A particular contribution is the clear and concise semantics of ABORT operations which arises naturally from this formalization. The section closes with a collection of lemmas describing useful properties of serial systems.

Next, a lock-based concurrent system is presented. Section 4 contains a description of a special type of object, called a "resilient object," which is used in the concurrent

system. Section 5 describes the remainder of the concurrent system, the "concurrent scheduler." This concurrent scheduler includes "lock manager" modules for all the objects; lock managers coordinate concurrent accesses.

Section 6 defines a system which is closely related to the concurrent system, the "weak concurrent system." This system preserves serial correctness for those transactions whose ancestors do not abort (i.e., those that are not "orphans"). Since the root of the transaction tree, T_0, has no ancestor, weak concurrent systems are correct for T_0. Section 7 contains complete proofs of correctness of the concurrent and weak concurrent systems; concurrent systems are serially correct, and weak concurrent systems are correct for T_0. The stronger condition is obtained for concurrent systems as a corollary to a result about weak concurrent systems.

It is interesting that the concurrent system algorithms are described in complete detail (essentially, in "pseudocode"), yet significant formal claims about their behavior can be stated clearly and easily. Although the full presentation involves a large number of lemmas, the ideas described by the lemmas are quite simple and intuitive. We think it is remarkable that these interesting properties of concurrent systems can be proved with complete rigor, in full detail, in so short a development. Despite the detailed level of presentation, the underlying model is general enough that the results apply to a wide range of implementations.

The style of the correctness proof is also noteworthy. It is a constructive proof, in that for each execution of the weak concurrent system and each non-orphan transaction, an execution of the serial system is explicitly constructed. The transaction's local "view" in the constructed execution is identical to that in the original weak concurrent execution, establishing the correctness of the weak concurrent system. One may think of the weak concurrent system as maintaining consistent, parallel "world views" within which concurrent siblings execute. As siblings return to their parent, these parallel worlds are "merged" to form a single consistent view. The locking policy prevents collisions between different views at the shared data. This intuition is strongly supported and clarified by the correctness proof, which constructs the parallel views as different serial schedules consistent with each sibling's local history. Lemmas illustrate how these serial schedules can be merged as siblings return or abort to their parent.

Section 8 contains a discussion of the relationship of this work to previous results, and Section 9 contains an indication of the work that lies ahead.

2. Basic model

In this section, we present the basic I/O automaton model, which is used to describe all components of our systems. This model consists of rather standard, possibly infinite-state, nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. This model is very similar to models used by Milner,

Hoare [13, 21] and others. There are a few differences: first, we find it important to classify operations of any automaton or system of automata as either "input" or "output" operations, of that automaton or system, and we treat these two cases differently. Also, we allow identification of arbitrary numbers of operations from different automata, rather than just pairwise identification as considered in [21].

This paper is not intended to develop the basic model. For the general theory of I/O automata, including a unified treatment of finite and infinite behavior, we refer the reader to [20]. In the present treatment of concurrent transaction systems, we only prove properties of finite behavior, so we only require a simple special case of the general model.

2.1. *I/O automata*

All components in our systems, transactions, objects and schedulers, will be modelled by *I/O automata.* An I/O automaton $\mathscr{A}$ has components *states*($\mathscr{A}$), *start*($\mathscr{A}$), *out*($\mathscr{A}$), *in*($\mathscr{A}$), and *steps*($\mathscr{A}$). Here, *states*($\mathscr{A}$) is a set of states, of which a subset *start*($\mathscr{A}$) is designated as the set of start states. The next two components are disjoint sets: *out*($\mathscr{A}$) is the set of *output operations*, and *in*($\mathscr{A}$) is the set of *input operations.* The union of these two sets is the set of *operations* of the automaton. Finally, *steps*($\mathscr{A}$) is the transition relation of $\mathscr{A}$, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an operation. This triple means that in state s', the automaton can atomically do operation π and change to state s. An element of the transition relation is called a *step* of $\mathscr{A}$.

The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. Our partitioning of operations into input and output indicates that each operation is only triggered in one place. We require the following condition.

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

This condition says that an I/O automaton must be prepared to receive any input operation at any time. This condition makes intuitive sense if we think of the input operations as being triggered externally. (In this paper, this condition serves mainly as a technical convenience, but in [20], where infinite behavior is considered, it is critical.)

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

Lemma 1. *If α is a schedule of I/O automaton $\mathscr{A}$, then every prefix of α is a schedule of $\mathscr{A}$.*

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

2.2. Composition of automata

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

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

The triple (s', π, s) is in the transition relation of $\mathscr{S}$ if and only if, for each component automaton $\mathscr{A}$, one of the following two conditions holds. Either π is an operation of $\mathscr{A}$, and the projection of the step onto $\mathscr{A}$ is a step of $\mathscr{A}$, or else π is not an operation of $\mathscr{A}$, and the states corresponding to $\mathscr{A}$ in the two tuples s' and s are identical. Thus, each operation of the composed automaton is an operation of a subset of the component automata. During an operation π of $\mathscr{S}$, each of the components which has operation π carries out the operation, while the remainder stay in the same state. Again, the operation π is an output operation of the composition if it is the output operation of a component—otherwise, π is an input operation of the composition.[1]

[1] Note that our model has chosen a particular convention for identifying operations of different components in a system: we simply identify those with the same name. This convention is simple, and sufficient for what we do in this paper. However, when this work is extended to more complicated systems, it may be expedient to generalize the convention for identifying operations, to permit reuse of the same operation name internally to different components. This will require introducing a renaming operator for operations, or else defining composition with respect to a designated equivalence relation on operations. We leave this for later work.

Lemma 2. *The composition of I/O automata is an I/O automaton.*

The next lemma allows us to compose automata in any order.

Lemma 3. *Up to isomorphism, composition of I/O automata is associative and commutative.*

An *execution* of a system is defined to be an execution of the automaton composed of the individual automata of the system. If α is a schedule of a system with component $\mathscr{A}$, then we denote by $\alpha|\mathscr{A}$ the subsequence of α containing all the operations of $\mathscr{A}$. Clearly, $\alpha|\mathscr{A}$ is a schedule of $\mathscr{A}$.

The next lemma expresses a "locality" property of I/O automata systems that greatly simplifies reasoning about such systems. Since only the component automaton $\mathscr{A}$ for which an operation π is an output can prevent the occurrence of that operation in a schedule of the system, it suffices to show that the operation is permitted by the local state of $\mathscr{A}$ in order to conclude that it is permitted by the global state of the system.

Lemma 4. *Let α' be a schedule of a system $\mathscr{S}$, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathscr{A}$. If $\alpha|\mathscr{A}$ is a schedule of $\mathscr{A}$, then α is a schedule of $\mathscr{S}$.*

Proof. Since $\alpha|\mathscr{A}$ is a schedule of $\mathscr{A}$, there is an execution β of $\mathscr{A}$ with schedule $\alpha|\mathscr{A}$. Let β' be the execution of $\mathscr{A}$ consisting of all but the last step of β. Similarly, since α' is a schedule of $\mathscr{S}$, there is an execution γ of $\mathscr{S}$ with schedule α'. It is possible that $\mathscr{A}$ has an execution in γ which is different from β' since different executions may have the same schedule. But it is easy to show, by induction on the length of γ, that there is another execution γ' of $\mathscr{S}$ in which component $\mathscr{A}$ has execution β', and which is otherwise identical to γ. The schedule of γ' is α'. Since π is not an output operation of any other component, π is defined from the state reached at the end of γ' so that $\alpha = \alpha'\pi$ is a schedule of $\mathscr{S}$. □

3. Serial systems

In this paper, we define three kinds of systems: "serial systems" and two types of "concurrent systems." Serial systems describe serial execution of transactions. Serial systems are defined for the purpose of providing a correctness condition for other systems: that the schedules of the other systems should "look like" schedules of the serial system to the transactions. As with serial schedules of single-level transaction systems, our serial schedules are too inefficient to use in practice. Thus, we define systems which allow concurrency, and which permit the abort of transactions after they have performed some work. We then prove that the schedules permitted by concurrent systems are correct.

In this section, we define "serial systems." Serial systems consist of "transactions" and "basic objects" communicating with a "serial scheduler." Transactions and basic objects describe user programs and data respectively. The serial scheduler controls communication between the other components, and thereby defines the allowable orders in which the transactions may take steps. All three types of system components are modelled as I/O automata.

We begin by defining a structure which describes the nesting of transactions. Namely, a *system type* is a four-tuple $(\mathcal{T}, \text{parent}, \mathcal{O}, V)$, where $\mathcal{T}$, the set of transaction names, is organized into a tree by the mapping parent: $\mathcal{T} \to \mathcal{T}$, with T_0 as the root. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) The leaves of this tree are called *accesses*. The set $\mathcal{O}$ denotes the set of objects; formally, $\mathcal{O}$ is a partition of the set of accesses, where each element of the partition contains the accesses to a particular object. The set V is a set of *values*, to be used as return values of transactions.

The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be an infinite structure.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of a "mythical" transaction, T_0, the root of the transaction tree. (In work on nested transactions, such as ARGUS [16, 18], the children of T_0 are often called "top-level" transactions.) It is very convenient to introduce the new root transaction to model the environment in which the rest of the transaction system runs. Transaction T_0 has operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in much the same way as about all of the other transactions, although it is distinguished from the other transactions by having no parent transaction. Since committing and aborting are operations which take place at the parent of each transaction (see below), T_0 can neither commit nor abort. Thus, a commit or abort of a top-level transaction is an irreversible step.

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

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

3.1. Transactions

This paper differs from earlier work such as [9, 19, 27] in that we model the transactions explicitly, as I/O automata. In modelling transactions, we consider it very important not to constrain them unnecessarily; thus, we do not want to require that they be expressible as programs in any particular high-level programming language. Modelling the transactions as I/O automata allows us to state exactly the properties that are needed, without introducing unnecessary restrictions or complicated semantics.

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

Input operations:

CREATE(T),
COMMIT(T', v) for $T' \in$ children(T) and $v \in V$,
ABORT(T') for $T' \in$ children(T).

Output operations:

REQUEST_CREATE(T') for $T' \in$ children(T),
REQUEST_COMMIT(T, v) for $v \in V$.

The CREATE input operation "wakes up" the transaction. The REQUEST_CREATE output operation is a request by T to create a particular child transaction.[2] The COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call COMMIT(T', v), for any v, and ABORT(T') *return* operations for transaction T'. The REQUEST_COMMIT operation is an announcement by T that is has finished its work, and includes a value recording the results of that work.

It is convenient to use two separate operations, REQUEST_CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST_CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. In actual systems such as ARGUS, this separation does occur, and the distinction will be important in our results and proofs. Similar remarks hold for the REQUEST_COMMIT and COMMIT operations.[3] We leave the

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

[3] Note that we do not include a REQUEST_ABORT operation for a transaction: we do not model the situation in which a transaction decides that its own existence is a mistake. Rather, we assign decisions to abort transactions to another component of the system, the scheduler. In practice, the scheduler must have some power to decide to abort transactions, as when it detects deadlocks or failures. In ARGUS, transactions are permitted to request to abort; we regard this request simply as a "hint" to the scheduler, to restrict its allowable executions in a particular way. This operation could be made explicit, constraining the scheduler to abort the requesting transaction, without substantively changing the model or results.

executions of particular transaction automata largely unspecified; the choice of which children to create, and what value to return, will depend on the particular implementation. For the purposes of the schedulers studied here, the transactions (and in large part, the objects) are "black boxes." Nevertheless, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints. Thus, transaction automata are required to preserve well-formedness, as defined below.

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

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

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

The following easy lemma summarizes the properties of well-formed sequences of transaction operations.

Lemma 5. *Let α be a well-formed sequence of operations of transaction T. Then the following conditions hold:*

(1) *The first operation of α is a* CREATE(T) *operation, and there are no other* CREATE *operations.*

(2) *If a* REQUEST_COMMIT *operation occurs in α, then there are no later output operations in α.*

(3) *There is at most one* REQUEST_CREATE(T') *operation for each child T' of T in α.*

(4) *Every return operation in α has a preceding* REQUEST_CREATE *operation in α for the same child transaction.*

3.2. Basic objects

Recall that I/O automata are associated with non-access transactions only. Since access transactions model abstract operations on shared data objects, we associate a single I/O automaton with each object, rather than one for each access. The operations for each object are just the CREATE and REQUEST_COMMIT operations for all the corresponding access transactions. Although we give these operations the same names as the operations of non-access transactions, it is helpful to think of the operations of access transactions in other terms also: a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation. Actually, these CREATE and REQUEST_COMMIT operations generalize the usual invocations and responses in that our operations carry with them a designation of the position of the access in the transaction tree. We depart from the traditional notational distinction between creation of subtransactions and invocations on objects since the common terminology for access and non-access transactions is of great benefit in unifying the statements and proofs of our results. Thus, a *basic object* X is modelled as an automaton, with the following operations.

Input operations:

CREATE(T) for T in accesses(X).

Output operations:

REQUEST_COMMIT(T, v) for T in accesses(X).

The CREATE operation is an invocation of an access to the object, while the REQUEST_COMMIT is a return of a value in response to such an invocation.

As with transactions, while specific objects are left largely unspecified, it is convenient to require that schedules of basic objects satisfy certain syntactic conditions. Thus, each basic object is required to preserve well-formedness, defined below.

Let α be a sequence of operations of basic object X. Then an access T to X is said to be *pending* in α provided that there is a CREATE(T), but no REQUEST_COMMIT for T, in α. We define *well-formedness* for finite sequences of operations of basic objects recursively. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of basic object X, where π is a single operation, then α

is well-formed provided that α' is well-formed, and the following hold:

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

These restrictions simply say that the same access does not get created more than once, nor does a creation of a new access occur at a basic object before the previous access has completed (i.e., requested to commit); also, a basic object does not respond more than once to any access, and only responds to accesses that have previously been created.

The following easy lemma summarizes the properties of well-formed sequences of basic object operations.

Lemma 6. *Let α be a well-formed sequence of operations of basic object X. Then α consists of alternating* CREATE *and* REQUEST_COMMIT *operations, starting with a* CREATE, *and with each consecutive* (CREATE, REQUEST_COMMIT) *pair having a common transaction.*

3.3. Serial scheduler

The third kind of component in a serial system is the serial scheduler. The serial scheduler is also modelled as an automaton. The transactions and basic objects have been specified to be any I/O automata whose operations and behavior satisfy simple syntactic restrictions. The serial scheduler, however, is a fully specified automaton, particular to each system type. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction after its parent has requested its creation, as long as the transaction has not actually been created. In the context of this scheduler, the "semantics" of an ABORT(T) operation are that transaction T was never created. The operations of the serial scheduler are as follows.

Input operations:

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

Output operations:

CREATE(T), COMMIT(T, v), ABORT(T).

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and object automata, and correspondingly for the CREATE, COMMIT and ABORT output operations. Each state s of the serial scheduler consists of five sets: create_requested(s), created(s), commit_requested(s), committed(s) and aborted(s). The set commit_requested(s) is a set of (transaction, value) pairs. The others are sets of transactions. We write returned(s) as an abbreviation for committed(s) $\cup$ aborted(s). There is exactly one initial state, in which the set create_requested is $\{T_0\}$, and the other sets are empty.

The transition relation consists of exactly those triples (s', π, s) satisfying the pre- and postconditions below, where π is the indicated operation. For brevity, we include in the postconditions only those conditions on the state s which may change with the operation. If a component of s is not mentioned in the postcondition (such as committed(s) in the postcondition for REQUEST_CREATE(T)), it is implicit that the set is the same in s' and s (that committed(s) = committed(s'), in this example). Note that here, as elsewhere, we have tried to specify the component as nondeterministically as possible, in order to achieve the greatest possible generality for our results.

- REQUEST_CREATE(T)
  Postcondition:

  $$\text{create\_requested}(s) = \text{create\_requested}(s') \cup \{T\}.$$

- REQUEST_COMMIT(T, v)
  Postcondition:

  $$\text{commit\_requested}(s) = \text{commit\_requested}(s') \cup \{(T, v)\}.$$

- CREATE(T)
  Precondition:

  $$T \in \text{create\_requested}(s') - \text{created}(s') - \text{aborted}(s'),$$

  $$\text{siblings}(T) \cap \text{created}(s') \subseteq \text{returned}(s');$$

  Postcondition:

  $$\text{created}(s) = \text{created}(s') \cup \{T\}.$$

- COMMIT(T, v)
  Precondition:

  $$(T, v) \in \text{commit\_requested}(s'), \qquad T \notin \text{returned}(s'),$$

  $$\text{children}(T) \cap \text{create\_requested}(s') \subseteq \text{returned}(s');$$

  Postcondition:

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

- ABORT(T)
  Precondition:

  $$T \in \text{create\_requested}(s') - \text{created}(s') - \text{aborted}(s'),$$

  $$\text{siblings}(T) \cap \text{created}(s') \subseteq \text{returned}(s');$$

  Postcondition:

  $$\text{aborted}(s) = \text{aborted}(s') \cup \{T\}.$$

The input operations, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded. A CREATE operation can only occur if a corresponding

REQUEST_CREATE has occurred and neither the CREATE nor a corresponding ABORT has already occurred. The second precondition on the CREATE operation says that the serial scheduler does not create a transaction until all its previously created sibling transactions have returned. That is, siblings are run sequentially. The precondition on the COMMIT operation says that the scheduler does not allow a transaction to commit to its parent until its children have returned. The precondition on the ABORT operation says that the scheduler does not abort a transaction while there is activity going on on behalf of any of its siblings. That is, aborted transactions are run sequentially with respect to their siblings. The next lemma relates a schedule of the serial scheduler to the state which results from applying that schedule.

Lemma 7. *Let α be a schedule of the serial scheduler, and let s be a state which can result from applying α to the initial state. Then the following conditions are true.*

(1) *T is in* create_requested(s) *exactly if $T = T_0$ or α contains a* REQUEST_CREATE(T) *operation.*
(2) *T is in* created(s) *exactly if α contains a* CREATE(T) *operation.*
(3) *(T, v) is in* commit_requested(s) *exactly if α contains a* REQUEST_COMMIT(T, v) *operation.*
(4) *T is in* committed(s) *exactly if α contains a* COMMIT *operation for T.*
(5) *T is in* aborted(s) *exactly if α contains an* ABORT(T) *operation.*
(6) *T is in* returned(s) *exactly if α contains a return operation for T.*

3.4. Serial systems and serial schedules

In this subsection, we define serial systems precisely and provide some useful terminology for talking about them.

The composition of transactions with basic objects and the serial scheduler for a given system type is called a *serial system.* Define the *serial operations* to be those operations which occur in the serial system: REQUEST_CREATES, REQUEST_COMMITS, CREATES, COMMITS and ABORTS. The schedules of a serial system are called *serial schedules.* The non-access transactions and basic objects are called the system *primitives.* (Recall that each basic object is an automaton corresponding to a set of access transactions. Thus, individual access transactions are not considered to be primitives.)

Recall that the operations of the basic objects have the same syntax as transaction operations. It is convenient to refer to CREATE(T) and REQUEST_COMMIT(T), when T is an access to basic object X, both as operations of transaction T and of object X. To avoid confusion, it is important to remember that there is no transaction automaton associated with any access operation.

For any serial operation π, we define *transaction*(π) to be the transaction at which the operation occurs. (For CREATE(T) operations and REQUEST_COMMIT operations for T, the transaction is T, while for REQUEST_CREATE(T) operations, and COMMIT and ABORT operations for T, the transaction is parent(T).) For a sequence α of serial operations, transaction(α) is the set of transactions of the operations in α.

Two sequences of serial operations, α and α', are said to be *equivalent* provided that they consist of the same operations, and $\alpha|P = \alpha'|P$ for each primitive P. Obviously, this yields an equivalence relation on sequences of serial operations.

We let $\alpha|T$ denote the subsequence of α consisting of operations whose transaction is T, even if T is an access. (This is an extension of the previous definition of $\alpha|T$, as accesses are not component automata of the serial system.)

Let α be a sequence of serial operations. We say that a transaction T is *live* in α provided that a CREATE(T), but no COMMIT(T, v) or ABORT(T), occurs in α. We say that transaction T' is *visible* to T in α provided that for each ancestor T'' of T' which is a proper descendant of lca(T, T'), some COMMIT(T'', v) occurs in α. (In particular, any ancestor of T is visible to T in α.) For sequence α and transaction T, let *visible*(α, T) be the subsequence of α consisting of operations whose transactions are visible to T in α. (These include access transactions T'.) We say that transaction T *sees everything* in α provided that visible(α, T) = α.

This is the same definition of visibility as appears, in a different model, in [19]. Visibility captures an intuitive notion suggested by the name: the transactions visible to a transaction T in α are those whose effects T is permitted to "see" in α.

If α is a sequence of operations, not necessarily all serial[4], then define *serial*(α) to be the subsequence of α consisting of the serial operations. We say that T is *live* in α provided that it is live in serial(α). We say that T' is *visible* to T in α if T' is visible to T in serial(α), and define visible(α, T) to be visible(serial(α), T). Also, T *sees everything* in α provided that T sees everything in serial(α). Similarly, define transaction(α) = transaction(serial(α)).

A finite sequence α of serial operations is said to be *well-formed* if its projection at every primitive is well-formed.

3.5. *Correctness condition*

We use serial schedules as the basis of our correctness definitions. Namely, we say that a finite sequence of operations is *serially correct for a primitive P* provided that its projection on P is identical to the projection on P of some serial schedule. We say that any finite sequence of operations is *serially correct* if it is serially correct for every non-access transaction. That is, α "looks like" a serial schedule to every non-access transaction.

In the remainder of this paper, we define two systems: concurrent systems and weak concurrent systems. We show that schedules of concurrent systems are serially correct, and that schedules of weak concurrent systems are serially correct for T_0.

Thus, we use the serial scheduler as a way of describing desirable behavior, just as serial schedules describe desirable behavior in more classical concurrency control settings (those without nesting). Then serial correctness plays the role in our theory that serializability plays in classical settings.

[4] We will introduce other kinds of operations later in the paper.

Motivation for our use of serial schedules to define correctness derives from the simple behavior of the serial scheduler, which determines the sequence of interactions between the primitives. Each transaction T is created only after parent(T) requests it, no siblings of T are created until T has returned, T is not committed until each of its requested children has itself returned, and T is not aborted until each of its created siblings has returned. The result is a depth-first traversal of the transaction tree, with requests flowing down and responses flowing up. We believe this depth-first traversal to be a natural notion of correctness which corresponds precisely to the intuition of how nested transaction systems ought to behave. Furthermore, it is a natural generalization of serializability, the correctness condition generally chosen for classical transaction systems.

Serial correctness is a condition which guarantees to implementors of transactions that their code will encounter only situations which can arise in serial executions. Correctness for T_0 is a natural alternative, which guarantees only that the external world will encounter situations which can arise in serial executions. This condition permits less constrained implementations, in that schedulers in such systems need not ensure that other transactions see consistent data. In particular, the weak concurrent scheduler presented below does not ensure correctness for transactions which have aborted ancestors (orphans). On the other hand, in such systems the authors of transactions must ensure that their programs behave well even if they see inconsistencies. (For example, transactions that see inconsistent data should not consume too many system resources, garble data beyond repair, dispense drugs or initiate military hostilities.) We hope this work will provide a tool for exploring the inherent costs of different correctness conditions such as these.

Note that our correctness conditions are defined at the transaction interface only, and do not constrain the object interface. We believe that this makes the conditions more meaningful to users, and more likely to suffice for a large variety of algorithms, which may use a variety of back-out, locking or version schemes to implement objects [5]. Previous work has focused on correctness conditions at the object interface [6, etc.]. While we believe that object interface conditions are important, their proper role in the theory is not to serve as the basic correctness condition. Rather, they are useful as intermediate conditions for proving correctness of particular implementations: such conditions can be shown to be sufficient, in combination with an appropriate scheduler, to ensure our correctness condition at the transaction interface. This observation is an important unifying contribution of our work. Our current research is focusing on demonstrating the usefulness of this approach, for a variety of object interface correctness conditions.

The serial correctness condition says that a schedule α must look like a serial schedule to each non-access transaction; this allows for the possibility that α might look like *different* serial schedules to different non-access transactions. This condition may at first seem to be too weak. It may seem that we should require that all transactions see a projection of the *same* serial schedule. But this stronger condition is not satisfied by most of the known concurrency control algorithms. It is true that

stronger conditions than ours can sometimes be proved, but such conditions are more complicated to state, and it is not yet clear which such conditions are most interesting.

The serial correctness condition is really not as weak as it may seem at first because T_0, the root transaction, is included among the transactions to which α must appear serial. As discussed above, transactions T_0 can be thought of as modelling the environment in which the rest of the transaction system runs. Its REQUEST_CREATE operations correspond to the invocation of top-level transactions, while its COMMIT and ABORT operations correspond to return values and external effects of those transactions. Since α's projection on T_0 must be serial, the environment of the transaction system will see only results that could arise in a serial execution. Indeed, this is the justification of the correctness condition for the weak concurrent system, whose schedules are shown to be correct for T_0, but not necessarily for any other transaction.

It is possible to use a different serial scheduler as a basis for correctness conditions. For example, the scheduler might delay creating one sibling until another *requests* to return, rather than until it actually returns to the parent [28]. Such a scheduler would provide less information to the parent about the actual order in which its children are executed, and consequently, provide more freedom for concurrent schedulers to schedule various events. Timestamp-based systems such as [23, 24] may support this weaker correctness condition, rather than the one described above, but this remains to be studied.

Our approach is really a general technique for studying operating system algorithms. A simple, intuitive and inefficient algorithm (automaton) is used to specify a "contract" between the users and implementor of an operating system. The user is guaranteed that applications (transactions, in our work) which are correct when run with the simple algorithm will also be correct when run with the actual operating system, which presumably will be more efficient. On the other hand, the implementor also has a formal and intuitive specification of the user interface.

3.6. Properties of serial systems

In this subsection, we prove a number of lemmas about the behavior of serial systems. They are collected here for reference later in this paper and in future work. Most of the lemmas describe properties that are quite easy to understand and believe, and the corresponding proofs are very straightforward. In the last paragraph of this subsection, there are some specialized lemmas that are somewhat more difficult. These are used in the proof of the main theorem in Section 7.

3.6.1. Fundamental properties of visibility

The first few lemmas give fundamental properties of visibility in sequences of serial operations. In this paragraph, we do not even require that the sequences be schedules of serial systems, but only that they be sequences of serial operations. The proofs of these lemmas are straightforward from the definitions.

Lemma 8. *Let α be a finite sequence of serial operations, and T, T' and T'' transactions.*

(1) *If T' is a descendant of T, then T is visible to T' in α.*
(2) *T' is visible to T in α if and only if T' is visible to* $\mathrm{lca}(T, T')$ *in α.*
(3) *If T'' is visible to T' in α and T' is visible to T in α, then T'' is visible to T in α.*
(4) *If T' is a descendant of T and T'' is visible to T in α, then T'' is visible to T' in α.*
(5) *If T' is a descendant of T and T' is visible to T'' in α, then T is visible to T'' in α.*
(6) *If T' is a proper descendant of T, T'' is visible to T' in α, but T'' is not visible to T in α, then T'' is a descendant of the child of T which is an ancestor of T'.*

Lemma 9. *Let α and β be sequences of serial operations, with β a subsequence of α.*

(1) *If transaction T is visible to transaction T' in β, then it is visible to transaction T' in α.*
(2) *If operation π is in* $\mathrm{visible}(\beta, T)$, *then it is in* $\mathrm{visible}(\alpha, T)$.

Lemma 10. *Let α, α', β and β' be sequences of serial operations, and let T and T' be transactions.*

(1) *If α is equivalent to α', and T' is visible to T in α, then T' is visible to T in α'.*
(2) *If α is equivalent to α', then* $\mathrm{visible}(\alpha, T)$ *is equivalent to* $\mathrm{visible}(\alpha', T)$.
(3) *If β is equivalent to β', then $\alpha - \beta = \alpha - \beta'$.*
(4) *If α is equivalent to α', and β is equivalent to β', then $\alpha - \beta$ is equivalent to $\alpha' - \beta'$.*
(5) *If* $\beta = \mathrm{visible}(\alpha, T)$, *then T sees everything in β.*
(6) *If β is equivalent to* $\mathrm{visible}(\alpha, T)$, *then T sees everything in β.*
(7) *If* $\beta = \mathrm{visible}(\alpha, T)$ *and T' is visible to T in α, then* $\mathrm{visible}(\beta, T') = \mathrm{visible}(\alpha, T')$.
(8) *If β is equivalent to* $\mathrm{visible}(\alpha, T)$, *$\beta'$ is equivalent to* $\mathrm{visible}(\alpha, T')$, *and T' is visible to T in α, then β' is equivalent to* $\mathrm{visible}(\beta, T')$.

Lemma 11. *Let α be a sequence of serial operations, and let T and T' be transactions. Then* $\mathrm{visible}(\alpha, T)|T'$ *is equal to $\alpha|T'$ if T' is visible to T in α, and is equal to the empty string otherwise.*

Lemma 12. *Let $\alpha\pi$ be a sequence of serial operations, where π is a single operation. Let T be a transaction and assume that* $\mathrm{transaction}(\pi)$ *is visible to T in $\alpha\pi$. Assume that π is not a* COMMIT *operation. Then* $\mathrm{visible}(\alpha\pi, T) = \mathrm{visible}(\alpha, T)\pi$.

3.6.2. Operations in serial schedules

The lemmas in this paragraph describe the kinds and orders of operations that can occur in well-formed serial schedules. In the next paragraph, we show that all serial schedules are well-formed, so that all these properties actually follow just from the fact that the schedules are serial.

Lemma 13. *Let α be a well-formed serial schedule, and let $T \neq T_0$ be a transaction.*

(1) *If α contains any operation with transaction T, then α contains a* REQUEST_CREATE(T).

(2) *If α contains a* COMMIT *for T, then α contains a* REQUEST_COMMIT *for T, a* CREATE(T) *and a* REQUEST_CREATE(T).

(3) *If α contains an* ABORT(T), *then α contains a* REQUEST_CREATE(T).

Proof. Straightforward from well-formedness and the scheduler preconditions. □

Lemma 14. *Let α be a well-formed serial schedule, and T a transaction. Assume that some descendant of T is in* transaction(α). *Then the following hold.*

(1) CREATE(T) *occurs in α.*

(2) *If $T \neq T_0$, then* REQUEST_CREATE(T) *occurs in α.*

Proof. (1): By induction on the length of α. The basis is easy. Let $\alpha = \alpha'\pi$, where π is a single operation, and assume that the result holds for α'. Let $T' =$ transaction(π), and let T be any ancestor of T'. We must show that CREATE(T) occurs in α.

Because α is well-formed, CREATE(T') occurs in α. If $T = T'$, we are done. Otherwise, Lemma 13 implies that REQUEST_CREATE(T') occurs in α. This occurs at parent(T'), which is a descendant of T. The inductive hypothesis then implies that α contains a CREATE(T).

(2): By part (1) and Lemma 13. □

Lemma 15. *Let α be a serial schedule, and let T be a transaction. Then α cannot contain both a* CREATE(T) *and an* ABORT(T) *operation.*

Proof. By the scheduler preconditions. □

Lemma 16. *Let α be a well-formed serial schedule, and let T be a transaction. If* ABORT(T) *occurs in α, then α contains no operations whose transactions are descendants of T.*

Proof. Assume the contrary. Then Lemma 14 implies that a CREATE(T) operation occurs in α. But Lemma 15 yields a contradiction. □

Lemma 17. *Let α be a well-formed serial schedule, and let $T \neq T_0$ be a transaction.*

(1) *If α contains a* REQUEST_CREATE(T), *but does not contain a return operation for T, then* parent(T) *is live in α.*

(2) *If T is live in α, then* parent(T) *is live in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T) *or an* ABORT(T), *then* parent(T) *is live in α.*

Proof. (1): Well-formedness implies that the REQUEST_CREATE(T) is preceded in α by a CREATE(parent(T)). Suppose that parent(T) is not live in α. Then a return operation for parent(T) occurs in α. By Lemma 15, ABORT(parent(T)) cannot appear in α. Thus, a COMMIT operation for parent(T) must appear in α. This COMMIT operation for parent(T) must be preceded by a REQUEST_COMMIT for parent(T), by the scheduler preconditions. By well-formedness, the REQUEST_COMMIT for parent(T) must follow the REQUEST_CREATE(T) operation, so that the COMMIT for parent(T) follows the REQUEST_CREATE(T) operation. Then by the scheduler preconditions for the COMMIT operation, there must be a return operation for T in α, a contradiction.

(2): Since T is live in α, CREATE(T) occurs in α and so Lemma 13 implies that REQUEST_CREATE(T) occurs in α. The result then follows from part (1).

(3): Since there is no CREATE(T) in α, there can be no REQUEST_COMMIT for T, by well-formedness. Then there can be no COMMIT for T, by the scheduler preconditions. The result follows from part (1). □

Lemma 18. *Let α be a well-formed serial schedule, and let T be a transaction.*

(1) *If α contains a* REQUEST_CREATE(T) *but does not contain a return operation for T, then any proper ancestor of T is live in α.*

(2) *If T is live in α, then any ancestor of T is live in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T) *or an* ABORT(T), *then any proper ancestor of T is live in α.*

Proof. By repeated use of Lemma 17. □

Lemma 19. *Let α be a well-formed serial schedule, and let T and T' be transactions with T' a descendant of T. Assume that there is a* COMMIT *operation for T in α.*

(1) *If a* REQUEST_CREATE(T') *occurs in α, then there is a return operation for T' in α.*

(2) *If T' is in* transaction(α), *then there is a* COMMIT *operation for T' in α.*

Proof. (1): By Lemma 18.

(2): Lemma 13 implies that REQUEST_CREATE(T') occurs in α. Part (1) then implies that there is a return operation for T' in α. Since T' is in transaction(α), Lemma 16 implies that there cannot be an ABORT(T') in α. Thus, there is a COMMIT for T' in α. □

Lemma 20. *Let α be a well-formed serial schedule. If a return operation for T is in α, then it follows all operations in α whose transaction is T.*

Proof. Lemma 16 implies the result if an ABORT(T) occurs in α. So assume that a COMMIT for T occurs in α. This must be preceded by a REQUEST_COMMIT for T, by scheduler preconditions. Well-formedness implies that the REQUEST_COMMIT is

preceded by a CREATE(T), and is not followed by any output operations of T. Thus, the only operations of T that could follow the REQUEST_COMMIT are return operations for children of T. Let T' be a child of T for which a return operation occurs in α. By scheduler preconditions, there is only one return operation for T' in α. By Lemma 13, α also contains a REQUEST_CREATE(T'). Since this is an output operation of T, it precedes the REQUEST_COMMIT for T, and hence precedes the COMMIT for T. Then the scheduler preconditions imply that the return operation for T' precedes the COMMIT for T. □

Lemma 21. *Let α be a well-formed serial schedule. If a return operation for T is in α, then it follows all operations in α whose transactions are descendants of T.*

Proof. Since a return operation for T occurs in α, we have $T \neq T_0$. Let T' be a descendant of T, and assume for the sake of obtaining a contradiction that an operation π with transaction(π) = T' occurs after the return for T in α. Let α' be the prefix of α preceding π.

Lemma 16 implies the result if an ABORT(T) occurs in α. So assume that a COMMIT for T occurs in α. By Lemma 13, α' contains a REQUEST_CREATE(T') operation. Then Lemma 19 implies that α' contains a return operation for T'. But then the well-formed schedule $\alpha'\pi$ contains a return for T' followed by an operation of T', which contradicts Lemma 20. □

Lemma 22. *Let α be a well-formed serial schedule. If T is a pending access in $\alpha|X$, then T is live in α.*

Proof. If T is a pending access in $\alpha|X$, then a CREATE(T) occurs in α, but no REQUEST_COMMIT for T occurs in α. Thus, by the scheduler preconditions, no COMMIT for T can occur in α. □

Lemma 23. *Let α be a well-formed serial schedule, and let T and T' be distinct transactions live in α. Then the following are true.*

(1) *T and T' are not siblings.*

(2) *Either T is an ancestor of T' or vice versa.*

Proof. (1): Assume the contrary. Assume without loss of generality that CREATE(T) precedes CREATE(T') in α. Then the scheduler preconditions for the CREATE(T') operation imply that a return operation for T occurs in α. This contradicts the assumption that T is live in α.

(2): By part (1) and Lemma 18. □

3.6.3. Well-formedness

Now we show that all serial schedules are well-formed. It follows that all the properties proved in the previous paragraph for well-formed serial schedules are

actually true for all serial schedules. Subsequently, we will use these properties without explicitly mentioning well-formedness.

Lemma 24. *Let α be a serial schedule. Then α is well-formed.*

Proof. By induction on the length of schedules. The base, length = 0, is trivial. Suppose that $\alpha\pi$ is a serial schedule, and assume that α is well-formed. If π is an output of a primitive P, then $\alpha\pi|P$ is well-formed because P preserves well-formedness, and so $\alpha\pi$ is well-formed. So assume that π is an input to a primitive P. It suffices to show that $\alpha\pi|P$ is well-formed. There are four cases.

(1) π is CREATE(T) and T is a non-access transaction. The scheduler preconditions insure that CREATE(T) does not appear in α.

(2) π is COMMIT(T, v) for some transaction T and value v. Then π is an input to transaction parent(T) = T'. The scheduler preconditions imply that α contains a REQUEST_COMMIT(T, v), and so Lemma 13 implies that α contains a REQUEST_CREATE(T). Also, the scheduler preconditions imply that no return operation for T occurs in α.

(3) π is ABORT(T) for some transaction T. Then π is an input to transaction parent(T) = T'. The scheduler preconditions imply that α contains a REQUEST_CREATE(T), but no return operation for T.

(4) π is CREATE(T) and T is an access to basic object X. By the scheduler preconditions, no CREATE(T) or ABORT(T) appears in α, but a REQUEST_CREATE(T) appears in α. Assume for the sake of deriving a contradiction that T' is a pending access in $\alpha|X$. Then Lemma 22 implies that T' is live in α. Also, Lemma 17 implies that parent(T) is live in α. Then Lemma 23 implies that one of T' or parent(T) is an ancestor of the other; since T and T' are both leaves of the transaction tree, the only possibility is that parent(T) is a proper ancestor of T'. Let T'' be the sibling of T which is an ancestor of T'. Then T'' is live in α, by Lemma 18. That is, there is a CREATE(T''), but no COMMIT for T'' in α. But this contradicts the scheduler preconditions for π. Therefore, there is no pending access in $\alpha|X$. □

3.6.4. Visibility and serial schedules

In this paragraph, we prove interesting lemmas about visibility in serial schedules.

Lemma 25. *Let α be a serial schedule, and π an operation in α. Then* transaction(π) *is visible in α to some transaction which is live in α.*

Proof. Let T = transaction(π). Since α is not empty, T_0 is live in α. Let T' be the least ancestor of T which is live in α. The proof is by induction on the distance from T' to T. If $T = T'$, the result is trivial. So assume that $T \neq T'$. Then COMMIT(T) is in α, and so T is visible to parent(T) in α. Lemma 13 implies that α contains a REQUEST_CREATE(T) operation, which occurs at parent(T). Then the inductive

hypothesis implies that parent(T) is visible to T'. Then T is visible to T' by Lemma 8. □

Lemma 26. (1) *Let α be a serial schedule, T a transaction and X an object. Then* visible(α, T)$|X$ *is a prefix of* $\alpha|X$.

(2) *Let α be a serial schedule, T a transaction and P a primitive. Then* visible(α, T)$|P$ *is a prefix of* $\alpha|P$.

Proof. (1): Let π and ϕ be operations in $\alpha|X$, with π preceding ϕ, and ϕ an operation in visible(α, T). Let α' be the prefix of α preceding ϕ. Let T' = transaction(ϕ) and T'' = transaction(π). Since ϕ is either a CREATE or a REQUEST_COMMIT for T', well-formedness of α implies that T' is live in $\alpha'\phi$. Thus, by Lemma 23, the only live transactions in $\alpha'\phi$ are ancestors of T'. By Lemma 25, T'' is visible to an ancestor of T' in $\alpha'\phi$, and hence in α. By Lemma 8, T'' is visible to T' in α. But T' is visible to T in α, by assumption. Lemma 8 then implies that T'' is visible to T in α, which gives the result.

(2): Immediate from Lemma 11 and part (1). □

Lemma 27. *Let α be a nonempty serial schedule. Let π be the last operation in α which is an output of the serial scheduler. Then* transaction(π) *sees everything in α.*

Proof. Let T = transaction(π). We first show that T is live in α. Either π is a CREATE(T) or else it is a return operation for a child T' of T. In the latter case, Lemma 14 implies that CREATE(T) also occurs in α. Thus, in either case, CREATE(T) occurs in α. Now, if a return operation for T occurs in α, Lemma 21 implies that it follows π, which is impossible. Thus, no return operation for T occurs in α. It follows that T is live in α.

Then Lemma 23 implies that the only other transactions that are live in α must be ancestors or descendants of T. We claim that no proper descendants of T are live in α. So assume for the sake of obtaining a contradiction that U is a proper descendant of T which is live in α. Then U is a descendant of a child V of T, and V is live in α, by Lemma 18. Let α' be the prefix of α preceding π. There are three cases.

(1) π is CREATE(T). Then Lemma 14 yields a contradiction.

(2) π is a COMMIT operation for T', a child of T. Then $T' \neq V$ since T' is not live in α. But T' and V are both live in α', which contradicts Lemma 23.

(3) π is an ABORT(T'), for child T' of T. Then $T' \neq V$ since T' is not live in α. But V is live in α'. But then the scheduler preconditions for π are not satisfied, a contradiction.

Thus, no descendants are live in α, so the only transactions that are live in α are ancestors of T. Now let ϕ be any operation in α. Lemma 25 implies that transaction(ϕ) is visible in α to some ancestor of T, and hence to T. □

Lemma 28. *Let α be a serial schedule, and T a transaction. Then* visible(α, T) *is a serial schedule.*

Proof. We proceed by induction on the length of α. The basis, length = 0, is trivial. Let $\alpha = \alpha'\pi$, where π is a single operation. Fix transaction T, and let T' = transaction(π). If T' is not visible to T in α, then visible(α, T) = visible(α', T), and the result is true by inductive hypothesis. So assume that T' is visible to T in α.

If π is an output operation of a primitive P, then visible(α, T)$|P$ is a prefix of $\alpha|P$, by Lemma 26, and thus is a schedule of P. By the inductive hypothesis, visible(α', T) is a serial schedule. Also, visible(α, T) = visible(α', T)π by Lemma 12. Then Lemma 4 shows that visible(α, T) is a serial schedule.

On the other hand, if π is an output operation of the scheduler, then Lemma 27 implies that T' sees everything in α. But since T' is visible to T in α, it follows that T sees everything in α. Thus, visible(α, T) = α, a serial schedule. □

3.6.5. Reordering and combining serial schedules

In this paragraph, we describe ways in which serial schedules can be modified and combined to yield other serial schedules. These lemmas are used in the proof of the main theorem, in Section 7.

Lemma 29. *Let α and α' be two equivalent serial schedules. If β is a sequence of serial operations such that $\alpha\beta$ is a serial schedule, then $\alpha'\beta$ is a serial schedule, and is equivalent to $\alpha\beta$.*

Proof. Equivalence is trivial. The fact that $\alpha'\beta$ is a serial schedule follows because the preconditions of the serial scheduler depend only upon the presence of previous operations, not their order. □

The next lemma says that any serial schedule can be transformed by moving all the operations visible to any particular transaction to the beginning of the schedule, and the result is another serial schedule. This lemma can be thought of as describing a kind of "canonical form" for a serial schedule, with respect to a particular transaction.

Lemma 30. *Let α be a serial schedule, and T any transaction. Let β =* visible(α, T). *Then $\beta(\alpha - \beta)$ is equivalent to α and is serial.*

Proof. Let $\alpha' = \beta(\alpha - \beta)$. If P is any primitive, then Lemma 26 implies that $\beta|P$ is a prefix of $\alpha|P$. Thus, α' is equivalent to α.

To show that α' is serial, we proceed by induction on its prefixes. By Lemma 28, β is serial, so we can use β as the basis. Let $\gamma\pi$ be a prefix of α', where π is a serial operation in $\alpha - \beta$ and γ is a serial schedule. If π is an output operation of a primitive P, then $\gamma\pi|P$ is a prefix of $\alpha'|P$, equaling $\alpha|P$ by equivalence, which is

a schedule of P. Then Lemma 4 shows that $\gamma\pi$ is a serial schedule. So assume that π is an output operation of the serial scheduler.

Let s be the state of the serial scheduler after γ. Let $\gamma'\pi$ be the prefix of α ending in π, and let s' be the state of the serial scheduler after γ'. Then π is enabled in s'. We must show that π is enabled in s. This suffices, by Lemma 4.

Since every operation in γ' is also in γ, it follows that each component set of s' is a subset of the corresponding set of s. There are three cases.

(1) π is CREATE(T') for some transaction T'. Then transaction(π) = T', and T' is not visible to T in α. Then $T' \in$ create_requested(s') $\subseteq$ create_requested(s). Also, it is easy to show that $T' \notin$ created(s) and $T' \notin$ aborted(s). Now let U be in siblings(T') $\cap$ created(s). If $U \in$ created(s'), then $U \in$ returned(s') since π is enabled in s', a subset of returned(s), as needed. So suppose that $U \notin$ created(s'). Then CREATE(U) occurs in β, so U is visible to T in α.

Since α contains both CREATE(T') and CREATE(U), Lemma 23 implies that α must contain a COMMIT for at least one of T' or U. If α contains a COMMIT for U, then it occurs in β, so $U \in$ returned(s). On the other hand, if α contains a COMMIT for T', then T' is visible to U in α, so Lemma 8 implies that T' is visible to T in α, a contradiction.

(2) π is COMMIT(T', v) for some transaction T' and value v. Then transaction(π) is parent(T'), which is not visible to T in α. Then (T', v) is in commit_requested(s') $\subseteq$ commit_requested(s). Also, it is easy to show that T' is not in returned(s). Now let U be in children(T') $\cap$ create_requested(s). Then there is a REQUEST_CREATE(U) in γ. This REQUEST_CREATE(U) occurs at T', which cannot be visible to T in α since parent(T') is not visible to T in α. Thus, the REQUEST_CREATE(U) does not occur in β, so it occurs in γ'. Since π is enabled in s', we have $U \in$ returned(s') $\subseteq$ returned(s).

(3) π is ABORT(T') for some transaction T'. Then transaction(π) = parent(T'), and parent(T') is not visible to T in α. Then $T' \in$ create_requested(s') $\subseteq$ create_requested(s). Also, it is easy to show that $T' \notin$ created(s) and $T' \notin$ aborted(s). Now let $U \in$ siblings(T') $\cap$ created(s). Then CREATE(U) occurs in γ. But CREATE(U) occurs at U, and U cannot be visible to T in α since parent(U) = parent(T') is not visible to T in α. Therefore, CREATE(U) does not occur in β, so it occurs in γ'. Then U is in siblings(T') $\cap$ created(s') $\subseteq$ returned(s') $\subseteq$ returned(s). □

The following lemma is an easy consequence of the preceding one.

Lemma 31. *Let α be a finite sequence of serial operations, and let T and T' be two transactions with T' visible to T in α. Let β and β' be serial schedules, such that β is equivalent to* visible(α, T) *and β' is equivalent to* visible(α, T'). *Then $\beta'' = \beta'(\beta - \beta')$ is equivalent to β and serial.*

Proof. Let γ = visible(β, T'). Then γ is serial by Lemma 28. Lemma 30 implies that $\gamma(\beta - \gamma)$ is equivalent to β and serial. Lemma 10 implies that β' is equivalent to γ,

and thus that $\beta - \gamma = \beta - \beta'$. Then Lemma 29 implies that β'' is equivalent to $\gamma(\beta - \gamma)$ and serial. Thus, β'' is equivalent to β and serial. □

The next two lemmas are used in the proof of Theorem 68. Each describes a way of "cutting and pasting" two serial schedules to yield a new serial schedule.

Lemma 32. *Let $\alpha\beta_1$COMMIT(T', u) and $\alpha\beta_2$ be two serial schedules, and T, T' and T'' three transactions such that the following conditions hold:*

(1) *T' is a child of T'' and T is a descendant of T'' but not of T',*
(2) *T' sees everything in $\alpha\beta_1$,*
(3) *T sees everything in $\alpha\beta_2$,*
(4) *$\alpha = \text{visible}(\alpha\beta_1, T'') = \text{visible}(\alpha\beta_2, T'')$ and*
(5) *no basic object has operations in both β_1 and β_2.*

Then $\alpha\beta_1$COMMIT$(T', u)\beta_2$ is a serial schedule.

Proof. Note first that if $T = T''$, then β_2 is empty and the result is trivial. So assume that $T \neq T''$. Then T is a descendant of a child U of T'', and $U \neq T'$.

Any operation in $\alpha\beta_1$ whose transaction is not a descendant of T', must be in visible$(\alpha\beta_1, T'')$ by Lemma 8. Similarly, any operation in $\alpha\beta_2$ whose transaction is not a descendant of U must be in visible$(\alpha\beta_2, T'')$. Thus, β_1 and β_2 contain only operations at descendants of T' and U respectively. Since T' and U are distinct siblings, and by assumption no objects have operations in both β_1 and β_2, it follows that no primitive has an operation occurring in both β_1 and β_2.

We proceed by induction on prefixes of $\alpha\beta_1$COMMIT$(T', u)\beta_2$. Let $\alpha'\phi$ be a prefix of $\alpha\beta_1$COMMIT$(T', u)\beta_2$, with α' a serial schedule and ϕ a serial operation. We use $\alpha'\phi = \alpha\beta_1$COMMIT$(T', u)$ as the basis since $\alpha\beta_1$COMMIT(T', u) is a serial schedule by assumption. So assume that $\alpha' = \alpha\beta_1$COMMIT$(T', u)\beta'$ for some sequence β'. There are two cases, depending on whether ϕ is an output of a primitive or of the serial scheduler.

Suppose that ϕ is an output operation of a primitive P. Then β_1COMMIT(T', v) contains no operations at P. Thus, $\alpha'\phi|P = \alpha\beta'\phi|P$, which is a prefix of $\alpha\beta_2|P$, which is a schedule of P since $\alpha\beta_2$ is a serial schedule. Thus, $\alpha'\phi|P$ is a schedule of P. The result follows by Lemma 4.

So suppose ϕ is an output of the serial scheduler. Then transaction$(\phi) = V$ for some descendant V of U. Let s be the state of the serial scheduler after α', and let s' be the state of the serial scheduler after $\alpha\beta'$. Then the following relationships hold between s and s'.

(1) $V \in$ create_requested$(s') -$ created$(s') -$ aborted(s')
iff $V \in$ create_requested$(s) -$ created$(s) -$ aborted(s).
(2) children$(V) \cap$ create_requested$(s') \subseteq$ returned(s')
iff children$(V) \cap$ create_requested$(s) \subseteq$ returned(s).
(3) $(V, v) \in$ commit_requested(s') iff $(V, v) \in$ commit_requested(s).

(4) $V \notin \text{returned}(s')$ iff $V \notin \text{returned}(s)$.
(5) $\text{siblings}(V) \cap \text{created}(s') \subseteq \text{returned}(s')$
iff $\text{siblings}(V) \cap \text{created}(s) \subseteq \text{returned}(s)$

Since the operations in β_1 are all at descendants of T', and those of β_2 are all at descendants of U, the first four biconditionals are immediate from Lemma 7. If V is a proper descendant of U, the last biconditional also follows. It remains to show that $\text{siblings}(U) \cap \text{created}(s') \subseteq \text{returned}(s')$ iff $\text{siblings}(U) \cap \text{created}(s) \subseteq \text{returned}(s)$. But any sibling of U created in $\alpha\beta'$ is created in α', and the only sibling of U created in α' and not $\alpha\beta'$ is T', and $T' \in \text{returned}(s)$. Thus, the claims are true.

Since ϕ is enabled in s', the claims above imply that ϕ is also enabled in s. The result follows. □

Lemma 33. *Let α ABORT(T') and $\alpha\beta$ be two serial schedules, and let T, T' and T'' be transactions such that the following conditions hold:*
(1) *T' is a child of T'' and T is a descendant of T'' but not of T',*
(2) *T sees everything in $\alpha\beta$, and*
(3) $\alpha = \text{visible}(\alpha, T'') = \text{visible}(\alpha\beta, T'')$.
Then α ABORT(T')β is a serial schedule.

Proof. Similar to, but somewhat simpler than, the proof of Lemma 32. □

4. Resilient objects

Having stated our correctness conditions, we are now ready to begin describing implementations and proving that they meet the requirements. This section and the next are devoted to the description of a concurrent system which permits the abort of transactions that have performed steps. An important component of a concurrent system is a new kind of object called a "resilient object". A resilient object is similar to a basic object, but it has the additional capability to undo operations of transactions that it discovers have aborted. Resilient objects have no capabilities for managing concurrency: rather, they assume that concurrency control is handled externally (by lock manager components of the scheduler). This section defines resilient objects and presents some of their properties. It also digresses slightly from the main development by describing and proving correct a particular implementation of resilient objects, which are constructed by keeping multiple copies of corresponding basic objects. The resilient object manages these copies as versions of the data object. Upon learning of an abort, the appropriate stored version is used in place of the current version.

4.1. Definitions

Resilient object $R(X)$ mimics the behavior of basic object X, but has two additional input operations, INFORM_COMMIT_AT(X)OF(T) and INFORM_ABORT_AT(X)OF(T) for every transaction T. Upon receiving an INFORM_

ABORT_AT(X)OF(T), $R(X)$ erases any effects of accesses which are descendants of T. This property is made formal as the "Resiliency Condition" below.

$R(X)$ has the following operations, which we call $R(X)$-*operations.*

Input operations:

CREATE(T), T an access to X,
INFORM_COMMIT_AT(X)OF(T), INFORM_ABORT_AT(X)OF(T).

Output operations:

REQUEST_COMMIT(T, v), T an access to X.

In order to describe well-formedness for resilient objects, we require a technical definition for the set of transactions which are *active* after a finite sequence of $R(X)$-operations. Roughly speaking, the transactions which are active are those on whose behalf the object has carried out some activity, but whose fate the object does not know.

The definition is recursive on the length of the sequence of $R(X)$ operations. Namely, only T_0 is active after the empty sequence. Let $\alpha = \beta\pi$, where π is a single operation, and let A and B denote the sets of active transactions after α and β, respectively. Then

$$A = \begin{cases} B \cup \{T\} & \text{if } \pi \text{ is CREATE}(T), \\ B & \text{if } \pi \text{ is a REQUEST\_COMMIT for } T, \\ (B - \{T\}) \cup \{\text{parent}(T)\} & \text{if } \pi \text{ is INFORM\_COMMIT\_AT}(X)\text{OF}(T) \text{ and } T \text{ is in } B, \\ B & \text{if } \pi \text{ is INFORM\_COMMIT\_AT}(X)\text{OF}(T) \text{ and } T \text{ is not in } B, \\ B - \text{descendants}(T) & \text{if } \pi \text{ is INFORM\_ABORT\_AT}(X)\text{OF}(T). \end{cases}$$

Now we define *well-formedness* for finite sequences of $R(X)$ operations. Again, the definition is recursive. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of $R(X)$-operations, then α is well-formed provided that α' is well-formed, and the following hold.

- If π is CREATE(T), then
  - (i) there is no CREATE(T) in α',
  - (ii) all the transactions which are active after α' are ancestors of T.
- If π is a REQUEST_COMMIT for T, then
  - (i) there is no REQUEST_COMMIT for T in α', and
  - (ii) T is active after α'.
- If π is INFORM_COMMIT_AT(X)OF(T), then
  - (i) there is no INFORM_ABORT_AT(X)OF(T) in α',
  - (ii) if T is an access to X, then a REQUEST_COMMIT for T occurs in α'.
- If π is INFORM_ABORT_AT(X)OF(T), then
  - (i) there is no INFORM_COMMIT_AT(X)OF(T) in α'.

An immediate consequence of these definitions is that the transactions active after any well-formed sequence of $R(X)$-operations α are a subset of the ancestors of a single active transaction, which we denote least(α).

Now we define an "undo" operator, which, when applied to a finite sequence of $R(X)$-operations, "undoes" the actions of transactions which are known to have aborted. Namely, for α a finite sequence of $R(X)$-operations, define *undo*(α) recursively as follows. Define undo(λ) = λ, where λ is the empty sequence. Let $\alpha = \beta\pi$, where π is a single operation. If π is a serial operation (a CREATE or a REQUEST_COMMIT), then undo(α) = undo(β)π. If π is INFORM_COMMIT_AT(X)OF(T), then undo(α) = undo(β). If π is INFORM_ABORT_AT(X)OF(T), then undo(α) is the result of eliminating, from undo(β), all operations whose transactions are descendants of T. Note that undo(α) contains only serial operations.

Let α be any finite sequence of $R(X)$-operations, and let π be an operation in α of the form INFORM_ABORT_AT(X)OF(T). Then the *scope* of π in α is the subsequence γ of α consisting of operations eliminated by π. That is, if $\beta\pi$ is a prefix of α, then the scope of π in α is the subsequence of undo(β) consisting of operations whose transactions are descendants of T.

Resiliency Condition. Resilient object $R(X)$ *satisfies the resiliency condition* if, for every well-formed schedule α of $R(X)$, undo(α) is a schedule of basic object X.

We require that resilient object $R(X)$ preserve well-formedness and satisfy the resiliency condition.

The resiliency condition is the correctness condition required by the concurrent schedulers at the object interface. The well-formedness requirement is a syntactic restriction, and the condition that undo(α) be a schedule of basic object X expresses the required semantic relationship between the resilient object and the basic object it incorporates. The important property which must be preserved is that the correctness condition at the resilient objects, together with the behavior of the concurrent scheduler, assures correctness at the transaction boundaries.

4.2. Properties of resilient objects

This subsection contains a collection of simple lemmas about the properties of well-formed sequences of $R(X)$ operations.

Lemma 34. *Let $\alpha\pi$ be a well-formed sequence of $R(X)$ operations, with π a single operation. The following are true*:

(1) *If π is a serial operation, then* transaction(π) *is active after $\alpha\pi$.*

(2) *If T is an access active after a prefix of α but not after α, then T is not active after $\alpha\pi$.*

(3) *If π is a* REQUEST_COMMIT *for T, then* CREATE(T) *is the last serial operation in α.*

Proof. (1): Immediate from the definition of active and well-formedness.

(2): Because T has no descendants, it can only become active when a CREATE(T) operation occurs, which can only happen once in a well-formed schedule.

(3): Suppose the last serial operation in α is ϕ, with $\phi \neq$ CREATE(T). Let transaction(ϕ) = T'. By well-formedness, $T \neq T'$. Also by well-formedness, T is active in α, so that CREATE(T) must occur in α, and so precedes ϕ. By part (1), T is active following CREATE(T) and after π, and T' is active following ϕ. But T cannot be active when ϕ occurs, by well-formedness, contradicting part (2) of this lemma. □

Lemma 35. *Let α be a well-formed sequence of $R(X)$ operations. Let T and T' be accesses to X with $T \neq T'$, and let π and ϕ be serial operations with transactions T and T' respectively. If π precedes ϕ in α, then, between π and ϕ, there is either an* INFORM_ABORT_AT(X) *for some ancestor of T, or else there are* INFORM_COMMIT_AT(X)OF(U) *operations for all ancestors U of T which are not ancestors of T', occurring in order from lowest to highest in the transaction tree ordering.*

Proof. By part (3) of Lemma 34 and well-formedness, we may assume that $\phi =$ CREATE(T'). Lemma 34 implies that T is active immediately after π. By well-formedness, before CREATE(T') can occur, it must be that all transactions that are active are ancestors of T'. There are only two ways in which this can happen. One possibility is that $R(X)$ first receives INFORM_COMMITs for all ancestors of T up to lca(T, T') in order from lowest to highest in the transaction tree ordering. The other possibility is that $R(X)$ first receives an INFORM_ABORT for an ancestor of T. □

Lemma 36. *Let $\alpha\pi$ be a well-formed sequence of $R(X)$ operations, with $\pi =$* INFORM_ABORT_AT(X)OF(T). *Then* undo($\alpha\pi$) *is a prefix of* undo(α).

Proof. Suppose not. Then there is a subsequence $\phi\psi$ of two operations in undo(α), such that ψ is in undo($\alpha\pi$) and ϕ is not. Clearly, ϕ and ψ are serial operations, transaction(ϕ) is a descendant of T and transaction(ψ) is not. Since ϕ is not in the scope of an INFORM_ABORT in α, by Lemma 35, there is an INFORM_COMMIT between ϕ and ψ for every proper descendant of lca(transaction(ϕ), transaction(ψ)) that is an ancestor of transaction(ϕ), including T. This contradicts the well-formedness of $\alpha\pi$. □

Lemma 37. *Let α be a well-formed sequence of $R(X)$ operations, and let T be any transaction active in α, other than T_0. Then* undo(α) *contains an operation ϕ at a descendant T' of T, which is followed in α by an* INFORM_COMMIT *for every ancestor of T' which is a proper descendant of T.*

Proof. The proof is by induction on α, with a trivial basis. Let $\alpha = \alpha'\pi$ such that the lemma is true for α' and that π is a single operation. Let T be a transaction active after α. There are four cases.

(1) Suppose π is CREATE(T''). Then undo(α) = undo(α')π. If $T \neq T''$, the result is immediate by the induction hypothesis since T is active after α'. If $T = T''$, then the lemma follows with $\pi = \phi$.

(2) If π is a REQUEST_COMMIT for a transaction T'', then undo(α) = undo(α')π and the same transactions are active in α and α'. The result is immediate.

(3) Suppose π is an INFORM_COMMIT for a transaction T''. Then undo(α) = undo(α')π. If T is active after α', the result is immediate. If T is not active after α', it follows that T = parent(T''). The result is immediate from the induction hypothesis.

(4) Suppose π is an INFORM_ABORT for a transaction U. Since T is active after α, it was active after α' and U is not an ancestor of T. Let ϕ be the transaction of transaction T' which follows from the inductive hypothesis applied to T and α'. Since α is well-formed and α' contains INFORM_COMMITs for every ancestor of T' up to T, U is not an ancestor of T'. It follows that ϕ is in undo(α) and the result holds. □

Lemma 38. *Let α be a well-formed sequence of $R(X)$ operations, and let* least(α) = T. *If* undo(α) *is nonempty, then it ends in an operation of a descendant of T.*

Proof. If $T = T_0$, the result is trivial, so assume otherwise. By the previous lemma, undo(α) contains an operation ϕ at a descendant of T. Without loss of generality, assume that ϕ is the last operation in undo(α) at a descendant of T. If any other operation π followed ϕ in undo(α), by Lemma 35, α would contain INFORM_COMMITs for every ancestor of transaction(ϕ) up to lca(transaction(ϕ), transaction(π)), which includes T. Then T is not active in α, a contradiction. □

Lemma 39. *Let $\alpha\pi$ be a well-formed sequence of $R(X)$ operations, with* π = INFORM_ABORT_AT(X)OF(T). *If T is not an ancestor of* least(α), *then* undo($\alpha\pi$) = undo(α).

Proof. Suppose that T is not an ancestor of least(α) and that undo($\alpha\pi$) $\neq$ undo(α). Then undo(α) contains a serial operation ϕ at a descendant T' of T. By Lemma 38, ϕ is followed in undo(α) by an operation at a descendent of least(α). By Lemma 35, α contains an INFORM_COMMIT for every ancestor least(α) up to lca(least(α), T'), which includes T, contradicting the well-formedness of $\alpha\pi$. □

We are now able to show that the undo operator preserves well-formedness.

Lemma 40. *If α is a well-formed sequence of $R(X)$-operations, then* undo(α) *is a well-formed sequence of X-operations.*

Proof. The proof is by induction on the length of α. The basis is trivial. Assume $\alpha = \alpha'\pi$, where π is a single operation, and undo(α') is a well-formed sequence of

X-operations. If π is an INFORM-ABORT or INFORM_COMMIT, undo(α) is a prefix of undo(α'), by Lemma 36, and the result is immediate.

If π is CREATE(T), then undo(α) = undo(α')π. By the well-formedness of α, CREATE(T) does not appear in α', and so not in undo(α'). Hence, (i) is satisfied. To see (ii), assume that CREATE(T') occurs in undo(α'), for access T'. Then Lemma 35 implies that INFORM_COMMIT_AT(X)OF(T') occurs after CREATE(T') in α. Then well-formedness (the precondition for INFORM_COMMIT_AT(X)OF(T')) implies that a REQUEST_COMMIT for T' occurs in α', and well-formedness also implies that the REQUEST_COMMIT for T' follows the CREATE(T'). Therefore, the REQUEST_COMMIT occurs in undo(α'), and so T' is not pending in undo(α'). Thus, (ii) is satisfied.

If π is a REQUEST_COMMIT for T, then again undo(α) = undo(α')π, and by the well-formedness of α,

(i) no REQUEST_COMMIT for T appears in α', and so not in undo(α'), and

(ii) T is active after α', and it follows that CREATE(T') occurs in undo(α'). □

4.3. Construction of a resilient object

In this subsection, we describe a construction of a resilient object $R(X)$ from a basic object X.

Recall that a resilient object X is distinguished from a basic object in that it has INFORM_ABORT and INFORM_COMMIT operations, a different definition of well-formedness, and satisfies the resiliency condition. The resilient object $R(X)$ is constructed from the states, transition function and operation labels of the basic object X. The resilient object $R(X)$ maintains a collection of "copies of X" (i.e., remembers states of X), one for each active transaction, with a particular current copy (corresponding to the least active transaction) to which CREATE operations are sent. When $R(X)$ receives an INFORM_ABORT, the appropriate stored copy becomes the current copy, thereby erasing the effects of the operations in the scope of the INFORM_ABORT.

The state of $R(X)$ consists of a pair (act, map), where act is a set of "active" transactions, and map is a function from act to states of basic object X. In the well-formed executions of $R(X)$ (defined below), act will always be a subset of the set of ancestors of one particular transaction in act, called least(act). (In case act has no least member (which, again, will not arise in executions with well-formed schedules), define least(act) arbitrarily.) The copy for least(act) is considered to be current. The initial states of $R(X)$ are those in which act = $\{T_0\}$ and map(T_0) is an initial state of the basic object X. In the following specification of the operations of $R(X)$, let (act', map') be the state of $R(X)$ prior to the operation, and (act, map) be the state of $R(X)$ after the operation.

- CREATE(T), T an access to X

  Postcondition:

  $$\text{act} = \text{act}' \cup \{T\},$$
  $$\text{map}(U) = \text{map}'(U) \quad \text{for all } U \in \text{act} - \{T\},$$

map(T) = s, where (map'(least(act')), CREATE(T), s) is in the transition relation of X.

- INFORM_ABORT_AT(X)OF(T)
  Postcondition:

  act = act' − descendants(T), map(U) = map'(U) for all $U \in$ act.

- INFORM_COMMIT_AT(X)OF(T)
  Postcondition:

  if $T \in$ act' **then**
  begin
  act = (act' − {T}) ∪ {parent(T)}
  map(U) = map'(U) for $U \in$ act − {parent(T)}
  map(parent(T)) = map'(T)
  end
  if $T \notin$ act' **then** act = act' and map = map'.

- REQUEST_COMMIT(T, v)
  Precondition:

  least(act') = T,
  (map'(T), REQUEST_COMMIT(T, v), s) is in the transition relation of X;

  Postcondition:

  act = act',
  map(U) = map'(U) for all $U \in$ act − {T},
  map(T) = s.

Now we prove that this implementation is a correct resilient object.

Lemma 41. *Let α be a well-formed schedule of $R(X)$ which can leave $R(X)$ in state* (act, map). *Then* act *coincides witht he set of transactions which are active after α.*

Proof. The proof is by induction on the length of α. The basis is trivial. Let $\alpha = \alpha'\pi$, where π is a single operation. There are four cases, depending on the type of operation π. Each is immediate from the definition of active and the implementation of $R(X)$. □

Lemma 42. *Let α be a well-formed schedule of $R(X)$ which can leave $R(X)$ in state* (act, map). *Then the following conditions hold.*

- undo(α) *is a schedule of basic object X which can leave X in state* map(least(act)), *and*

- *if T' is any transaction other than T_0, and* αINFORM_ABORT_AT(X)OF(T')) *is well-formed, then* undo(αINFORM_ABORT_AT(X)OF(T')) *is a schedule of basic object X which can leave X in state* map(U), *where U is the least element of* act *which is not a descendant of T'.*

Proof. First, observe that if T' is not an ancestor of least(act), and αINFORM_ABORT_AT(X)OF(T') is well-formed, then Lemmas 41 and 39 imply that undo(αINFORM_ABORT_AT(X)OF(T')) = undo(α), so the second condition follows from the first.

The proof is by induction on the length of α. In each case, we prove the first condition, then the second condition assuming that T' is an ancestor of least(act). By the observation above, this is sufficient.

The basis is trivial. Let $\alpha = \alpha'\pi$, where π is a single operation. Let (act', map') be a state of $R(X)$ after α', such that ((act', map'), π, (act, map)) is a transition for $R(X)$. There are four cases.

(1) π = CREATE(T). Then undo(α) = undo(α')π. By the inductive assumption, undo(α') is a schedule of X which can leave X in state map'(least(act')). By the implementation of $R(X)$, (map'(least(act')), π, map(T)) is a transition of X, and T = least(act). Thus the first condition of the lemma is satisfied.

To see that the second condition holds, note that all active transactions after α are ancestors of T and, by well-formedness, are exactly the transactions active after α', together with T. Let ϕ be INFORM_ABORT_AT(X)OF(T'), where T' is an ancestor of T other than T_0, and $\alpha\phi$ is well-formed. If T' is a proper descendant of least(act'), by Lemma 39, undo($\alpha\phi$) = undo(α'), which is a schedule of basic object X which can leave X in state map(least(act')), by the inductive hypothesis. If T' is an ancestor of least(act'), undo($\alpha\phi$) = undo($\alpha'\phi$), the least element of act which is not a descendant of T' is also the least element of act' which is not a descendant of T', and the result follows by the inductive hypothesis.

(2) π = REQUEST_COMMIT(T, v). Then undo(α) = undo(α')π. By the inductive assumption, undo(α') is a schedule of X which can leave X in state map'(least(act')). By the implementation of $R(X)$, (map'(least(act')), π, map(T)) is a transition of X, and T = least(act). Thus the first condition of the lemma is satisfied.

To see that the second condition holds, note that the active transactions after α are all ancestors of T and, by well-formedness, are exactly the transactions active after α'. Let ϕ be INFORM_ABORT_AT(X)OF(T'), where T' is an ancestor of T other than T_0, and $\alpha\phi$ is well-formed. Then undo($\alpha\phi$) = undo($\alpha'\phi$), which is a schedule of basic object X which can leave X in state map(least(act')), by the inductive hypothesis. Furthermore, the least element of act which is not a descendant of T' is also the least element of act' which is not a descendant of T', and the result follows by the inductive hypothesis.

(3) π = INFORM_COMMIT_AT(X)OF(T). Then undo(α) = undo(α'). Also, map(least(act)) = map(least(act')), by definition of $R(X)$. The first condition follows.

By the definition of $R(X)$, least(act) is an ancestor of least(act'). Let ϕ be INFORM_ABORT_AT(X)OF(T'), where T' is an ancestor of least(act) other than T_0, and $\alpha\phi$ is well-formed. Then $\alpha'\phi$ is well-formed, and undo($\alpha\phi$) = undo($\alpha'\phi$). Also, since $\alpha\phi$ is well-formed, $T' \neq T$. Let U and U' be the least elements of act and act' respectively, which are not descendants of T'.

If $T \notin$ act', or if $U \neq$ parent(T), then $U = U'$ and map(U) = map'(U'), and the second condition follows from the inductive hypothesis. So assume that $T \in$ act' and U' = parent(T). Then since $T' \neq T$, it follows that $U' = T$. Then map'(U') = map(U), and the second condition again follows from the inductive hypothesis.

(4) π = INFORM_ABORT_AT(X)OF(T). If T is not an ancestor of least(act'), then undo(α) = undo(α'), by Lemma 39. Furthermore, the state of $R(X)$ is not changed, αINFORM_ABORT_AT(X)OF(T') is well-formed only if α'INFORM_ABORT_AT(X)OF(T') is, and the active transactions after α are exactly those active after α'. The result follows.

Suppose that T is an ancestor of least(act'). The first condition is immediate from the inductive hypothesis. Let ϕ be INFORM_ABORT_AT(X)OF(T'), where T' is an ancestor of least(act) other than T_0, and $\alpha\phi$ is well-formed. Since act = act' − descendants(T), least(act), and hence T', is an ancestor of T, undo($\alpha\phi$) = undo($\alpha'\pi\phi$) = undo($\alpha'\phi$), and the second condition follows as well. □

Theorem 43. *$R(X)$ is a resilient object.*

Proof. We must show that $R(X)$ preserves well-formedness and satisfies the resiliency condition. That $R(X)$ satisfies the resiliency condition follows immediately from Lemma 42.

Assume that α is a well-formed schedule of $R(X)$ and π is an output operation of $R(X)$ enabled after an execution with schedule α. We must show that $\alpha\pi$ is a well-formed sequence of $R(X)$-operations.

Since π is an output, it has the form REQUEST_COMMIT(T, v) for some access T and value v. Let (act, map) be a state of $R(X)$ after α such that π is enabled in (act, map). Clearly, π is an output of basic object X enabled from state map(least(act)). By Lemma 42, undo(α) is a schedule of basic object X which can leave X in state map(least(act)), so undo(α)π = undo($\alpha\pi$) is a schedule of basic object X.

Since X preserves well-formedness for basic objects and, by Lemma 40, undo(α) is a well-formed sequence of X-operations, undo(α) ends with the operation ϕ = CREATE(T) and contains no other operations with transaction T. Let $\beta\phi$ be the prefix of α ending in ϕ. Suppose first that a REQUEST_COMMIT for T occurs in α. Since α is well-formed, ϕ is the only CREATE(T) operation in α and, by Lemma 34, the second REQUEST_CREATE for T follows ϕ and, by the definition of undo, is in undo(α) if ϕ is a contradiction.

It remains to show that T is active after α. By Lemma 34, T is active after $\beta\phi$. No INFORM_COMMIT for T can occur after ϕ in α since, by well-formedness, there

is no REQUEST_COMMIT for T in α. Also, since ϕ is in undo(α), no INFORM_ABORT for an ancestor of T can occur after ϕ in α. Thus, T is still active after α. □

5. Concurrent systems

As with serial schedules in classical settings, our serial schedules contain no concurrency or resiliency and thus are too inefficient to use in practice. Their importance is solely for defining correctness for transaction systems. In this section, we define a new kind of system called a *concurrent system.* The new system consists of the same transactions as in a serial system, a resilient object $R(X)$ for every basic object X of the serial system, and a concurrent scheduler.

Concurrent systems describe computations in which transactions run concurrently and can be aborted after they have performed some work. The concurrent scheduler has the joint responsibility of controlling concurrency and of seeing that the effects of aborted transactions (and their descendants) become undone. Concurrent systems make use of the roll-back capabilities of resilient objects to make sure that ABORT operations in concurrent systems have the same semantics (so far as the transactions can tell) as they do in serial systems.

Concurrent systems are defined in this section. In the next section, the more permissive "weak concurrent systems" are defined. In Section 7, we prove that the schedules of concurrent systems are serially correct, as a corollary of a weaker correctness property for the weak concurrent system.

5.1. Lock managers

The scheduler we define is called the *concurrent scheduler.* It is composed of several automata: a *lock manager* for every object X, and a single *concurrent controller.* The job of the lock managers is to ensure that the associated object receives no CREATES until the lock manager has received abort or commit information for all necessary preceding transactions. This lock manager models an exclusive locking protocol derived from Moss' algorithm [22]. The lock manager has the following operations:

Input operations:

INTERNAL_CREATE(T), where T is an access to X,
INFORM_COMMIT_AT(X)OF(T) for T any transaction,
INFORM_ABORT_AT(X)OF(T) for T any transaction.

Output operations:

CREATE(T), where T is an access to X.

The input operations INTERNAL_CREATE, INFORM_COMMIT and INFORM_ABORT will compose with corresponding output operations of the concurrent scheduler which we will construct in this subsection. The output CREATE operation composes

with the CREATE input operation of the resilient object $R(X)$. The lock manager receives and manages requests to access object X, using a hierarchical locking scheme. It uses information about the commit and abort of transactions to decide when to release locks.

Each state s of the lock manager consists of the following three sets of transactions: lock_holders(s), create_requested(s), and created(s). Initially, lock_holders = $\{T_0\}$, and the other sets are empty. The operations work as follows.

- INTERNAL_CREATE(T)
  Postcondition:

  create_requested(s) = create_requested(s') $\cup \{T\}$.

- INFORM_COMMIT_AT(X)OF(T)
  Postcondition:

  if $T \in$ lock_holders(s')
  then lock_holders(s) = (lock_holders(s') $- \{T\}$) $\cup$ {parent(T)}.

- INFORM_ABORT_AT(X)OF(T)
  Postcondition:

  lock_holders(s) = lock_holders(s') $-$ descendants(T).

- CREATE(T)
  Precondition:

  $T \in$ create_requested(s') $-$ created(s'),
  lock_holders(s') $\subseteq$ ancestors(T);

  Postcondition:

  lock_holders(s) = lock_holders(s') $\cup \{T\}$,
  created(s) = created(s') $\cup \{T\}$.

Note that resilient object $R(X)$ and the lock manager for X share the INFORM_ABORT and INFORM_COMMIT input operations. These compose with the output from the concurrent controller defined below.

Thus, the lock manager only sends a CREATE(T) operation on to the object in case all the current lock_holders are ancestors of T. When the lock manager learns about the commit of a transaction T for which it holds a lock, it releases the lock to T's parent. When the lock manager learns about the abort of a transaction T for which it holds a lock, it simply releases all locks held by that transaction and its descendants. Our model provides an exceptionally simple and clear way of describing this important algorithm.

A key property of lock managers is described by the following lemma.

Lemma 44. *Let X be an object and let T and T' be accesses to X. Let U be an ancestor of T which is not an ancestor of T'. Let α be a schedule of the lock manager for X. If*

CREATE(T) *precedes* CREATE(T') *in* α, *then between the two* CREATE *operations, there is either an* INFORM_COMMIT_AT(X)OF(U) *operation, or else an* INFORM_ABORT_AT(X) *for some ancestor of* T.

Proof. At the time the CREATE(T) occurs, the lock manager puts T into the set of lock_holders. Before the lock manager can send in CREATE(T'), it must be that all the transactions in lock_holders are ancestors of T'. There are only two ways in which this can happen. One possibility is that the lock manager first receives INFORM_COMMITS for all ancestors of T up to lca(T, T'), including INFORM_COMMIT_AT(X)OF(U). The other possibility is that the lock manager first receives an INFORM_ABORT for an ancestor of T. □

5.2. The concurrent controller

The concurrent controller is similar to the serial scheduler, but it allows siblings to proceed concurrently. In order to manage this properly, it interacts with "concurrent objects" (lock managers and resilient objects) instead of just basic objects. The operations are as follows.

Input operations:

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

Output operations:

CREATE(T), T a non-access transaction,
INTERNAL_CREATE(T), T an access transaction,
COMMIT(T, v), ABORT(T),
INFORM_COMMIT_AT(X)OF(T),
INFORM_ABORT_AT(X)OE(T).

Each state s of the concurrent controller consists of five sets: create_requested(s), created(s), commit_requested(s), committed(s), and aborted(s). The set commit_requested(s) is a set of (transaction, value) pairs, and the others are sets of transactions. (As before, we will occasionally write $T \in$ commit_requested(s) for $(T, v) \in$ commit_requested(s) for some v.) All sets are initially empty except for create_requested, which is $\{T_0\}$. Define returned(s) = committed(s) $\cup$ aborted(s). The operations are as follows.

- REQUEST_CREATE(T)
  Postcondition:

  create_requested(s) = create_requested(s') $\cup \{T\}$.

- REQUEST_COMMIT(T, v)
  Postcondition:

  commit_requested(s) = commit_requested(s') $\cup \{(T, v)\}$.

- CREATE(T), T a non-access transaction
  Precondition:

  $T \in \text{create\_requested}(s') - \text{created}(s') - \text{aborted}(s')$;

  Postcondition:

  $\text{created}(s) = \text{created}(s') \cup \{T\}$.

- INTERNAL_CREATE(T), T an access transaction
  Precondition:

  $T \in \text{create\_requested}(s') - \text{created}(s') - \text{aborted}(s')$;

  Postcondition:

  $\text{created}(s) = \text{created}(s') \cup \{T\}$.

- COMMIT(T, v)
  Precondition:

  $(T, v) \in \text{commit\_requested}(s')$, $T \notin \text{returned}(s')$,
  $\text{children}(T) \cap \text{create\_requested}(s') \subseteq \text{returned}(s')$;

  Postcondition:

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

- ABORT(T)
  Precondition:

  $T \in (\text{create-requested}(s') - \text{created}(s') - \text{aborted}(s'))$
  $\cup (\text{commit\_requested}(s') - \text{returned}(s'))$,
  $\text{children}(T) \cap \text{create\_requested}(s') \subseteq \text{returned}(s')$;

  Postcondition:

  $\text{aborted}(s) = \text{aborted}(s') \cup \{T\}$.

- INFORM_COMMIT_AT(X)OF(T):
  Precondition:

  $T \in \text{committed}(s')$.

- INFORM_ABORT_AT(X)OF(T):
  Precondition:

  $T \in \text{aborted}(s')$.

The concurrent controller is closely related to the serial scheduler. In place of the serial scheduler's CREATE operations, the concurrent controller has two kinds of operations, CREATE operations and INTERNAL_CREATE operations. The former is used for interaction with non-access transactions, while the latter is used for interaction with access transactions. From the concurrent controller's viewpoint,

the two operations are the same; however, our naming convention for operations requires us to assign them different names since the INTERNAL_CREATE operations are intended to be identified with INTERNAL_CREATE operations of the lock managers (which also have CREATE operations for interaction with the resilient objects). The precondition on the serial scheduler's CREATE operation, which ensures serial processing of sibling transactions, does not appear in the concurrent controller. Thus, the concurrent controller may run any number of sibling transactions concurrently, provided their parent has requested their creation.

The concurrent controller's COMMIT operation is the same as the serial scheduler's COMMIT operation. The concurrent controller's ABORT operation is different, however; in addition to aborting a transaction in the way that the serial scheduler does, the concurrent controller has the additional capability to abort a transaction that has actually been created and has carried out some steps. In this particular formulation, aborts occur if the transaction was not created (as with the serial scheduler), or if the transaction has previously requested to commit, and its children have returned. Together with the requirements on the COMMIT operation, this condition ensures that all transaction completion occurs bottom-up. In the weak concurrent system to be considered in Section 6, a different, "weak," concurrent controller will be used; it differs from the concurrent controller of this section precisely in not requiring ABORT operations to wait for their transactions (and subtransactions) to complete.

The concurrent controller also has two additional operations not present in the serial scheduler. These operations allow the concurrent controller to forward necessary abort and commit information to the lock managers and resilient objects.

Lemma 45. *Let α be a schedule of the concurrent scheduler, and let s be a state which can result from applying α to the initial state. Then the following conditions are true.*

(1) *T is in* create_requested(s) *exactly if $T = T_0$ or α contains a* REQUEST_CREATE(T) *operation.*

(2) *If T is a non-access transaction, then T is in* created(s) *exactly if α contains a* CREATE(T) *operation.*

(3) *If T is an access transaction, then T is in* created(s) *exactly if α contains an* INTERNAL_CREATE(T) *operation.*

(4) *(T, v) is in* commit_requested(s) *exactly if α contains a* COMMIT_REQUEST(T, v) *operation.*

(5) *(T, v) is in* committed(s) *exactly if α contains a* COMMIT(T, v) *operation.*

(6) *T is in* aborted(s) *exactly if α contains an* ABORT(T) *operation.*

5.3. Concurrent systems

The composition of transactions, resilient objects and the concurrent scheduler (lock managers and concurrent controller) is the *concurrent system.* A schedule of the concurrent system is a *concurrent schedule,* and the operations of a concurrent system are *concurrent operations.*

A finite sequence α of concurrent operations is *well-formed* if, for every primitive P, $\alpha|P$ is well-formed (using the appropriate definition of well-formedness).

The main result of this paper is that every concurrent schedule is serially correct. This will be proved as a corollary to a stronger result in Section 7.

5.4. Properties of concurrent systems

As we did for serial schedules, we now prove some useful basic properties for concurrent schedules. These lemmas describe the possible kinds and orders of operations that can occur in well-formed concurrent schedules. Later, we will see that all concurrent schedules are well-formed, so these properties actually follow just from the fact that these schedules are concurrent. All results and proofs in this subsection are straightforward.

Lemma 46. *Let α be a well-formed concurrent schedule, and let $T \neq T_0$ be a transaction.*

(1) *If α contains any operation with transaction T, then α contains a* CREATE(T) *and a* REQUEST_CREATE(T).

(2) *If α contains a* COMMIT *for T, then α contains a* REQUEST_COMMIT *for T, a* CREATE(T) *and a* REQUEST_CREATE(T).

(3) *If α contains an* ABORT(T), *then α contains a* REQUEST_CREATE(T).

Lemma 47. *Let α be a well-formed concurrent schedule, and T a transaction. Assume that some descendant of T is in* transaction(α). *Then the following hold.*

(1) CREATE(T) *occurs in α.*

(2) *If $T \neq T_0$, then* REQUEST_CREATE(T) *occurs in α.*

Lemma 48. *Let α be a well-formed concurrent schedule, and let $T \neq T_0$ be a transaction.*

(1) *If α contains a* REQUEST_CREATE(T), *but does not contain a return operation for T, then* parent(T) *is live in α.*

(2) *If T is live in α, then* parent(T) *is live in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T) *or* ABORT(T), *then* parent(T) *is live in α.*

Proof. (1): Well-formedness implies that the REQUEST_CREATE(T) is preceded by a CREATE(parent(T)). Suppose that parent(T) is not live in α. Then a return operation for parent(T) occurs in α. In case the return operation for parent(T) is an ABORT(parent(T)), scheduler preconditions imply that the CREATE(parent(T)) must precede the ABORT(parent(T)). Then the scheduler preconditions for the return operation imply that the return for parent(T) must be preceded by a REQUEST_COMMIT for parent(T). By well-formedness, the REQUEST_COMMIT for parent(T) must follow the REQUEST_CREATE(T), so that the return for parent(T) must follow the REQUEST_CREATE(T). Then the scheduler preconditions for the return operation imply that there must be a return operation for T in α, a contradiction.

(2) and (3) are as in Lemma 17. □

Lemma 49. *Let α be a well-formed concurrent schedule, and let T be a transaction.*

(1) *If α contains a* REQUEST_CREATE(T), *but does not contain a return operation for T, then all proper ancestors of T are live in α.*

(2) *If T is live in α, then all ancestors of T are live in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T) *or* ABORT(T), *then all proper ancestors of T are live in α.*

Lemma 50. *Let α be a well-formed concurrent schedule, and let T and T' be transactions with T' a descendant of T. Assume that there is a return operation for T in α.*

(1) *If there is a* REQUEST_CREATE(T') *in α, then there is a return operation for T' in α.*

(2) *If T' is in* transaction(α), *then there is a return operation for T' in α.*

Proof. (1): By Lemma 49.

(2): By Lemma 46 and part (1). □

Lemma 51. *Let α be a well-formed concurrent schedule. If a return operation for T is in α, then it follows all operations in α whose transaction is T.*

Proof. First consider the case where T is not an access. If no CREATE(T) occurs in α, the result is immediate, so assume that CREATE(T) occurs in α. In case an ABORT(T) occurs in α, scheduler preconditions imply that the CREATE(T) must precede the ABORT(T). Then the return operation for T must be preceded by a REQUEST_COMMIT for T, by scheduler preconditions. Well-formedness implies that the REQUEST_COMMIT is preceded by CREATE(T), and is not followed by any output operations of T. Thus, the only serial operations of T that could follow the REQUEST_COMMIT are return operations of children of T.

Let T' be a child of T for which a return operation occurs in α. By scheduler preconditions, there is only one return operation for T' in α. By Lemma 46, α also contains REQUEST_CREATE(T'). Since this is an output operation of T, it precedes the REQUEST_COMMIT for T, and hence precedes the return operation for T. Then the scheduler preconditions imply that the return operation for T' precedes the return for T.

Next, consider the case where T is an access. If no INTERNAL_CREATE(T) occurs in α, the result is immediate, so assume that INTERNAL_CREATE(T) occurs in α. In case an ABORT(T) occurs in α, scheduler preconditions imply that the INTERNAL_CREATE(T) must precede the ABORT(T). Then the return operation for T must be preceded by a REQUEST_COMMIT for T, and well-formedness implies that this is in turn preceded by CREATE(T). Thus, no serial operations of T can follow the return operation for T. □

Lemma 52. *Let α be a well-formed concurrent schedule. If a return operation for T is in α, then it follows all operations in α whose transactions are descendants of T.*

Proof. Since a return operation for T occurs in α, we have $T \neq T_0$. Let T' be a descendant of T, and assume for the sake of obtaining a contradiction that a serial operation π with transaction(π) = T' occurs after the return for T in α. Let α' be the prefix of α preceding π.

By Lemma 46, α' contains a REQUEST_CREATE(T'). Then Lemma 50 implies that α' must contain a return operation for T'. But then the well-formed schedule $\alpha'\pi$ contains a return operation for T' followed by an operation of T', which contradicts Lemma 51. □

Weak concurrent systems are defined in the following section, and many of their properties are stated and proved. Weak concurrent systems are obtained by replacing the concurrent scheduler with a more permissive scheduler, the weak concurrent scheduler. Results in Section 7 prove that every execution of the concurrent system is also an execution of the weak concurrent system. Thus, additional interesting properties of concurrent system behavior follow immediately from the corresponding properties of weak concurrent system behavior, proved in that section.

6. Weak concurrent systems

In this section, we define "weak concurrent systems", which are exactly the same as concurrent systems, except that they have a more permissive controller, the "weak concurrent controller". The weak concurrent controller reports aborts to a transaction's parent while there is still activity going on in the aborted transaction's subtree. In this paper, weak concurrent systems are used primarily to provide an intermediate step in proving the correctness of concurrent systems: proving a weaker condition for weak concurrent systems allows us to infer the stronger correctness condition for concurrent systems. However, weak concurrent systems are also of interest in themselves. In a distributed implementation of a nested transaction system, performance considerations may make it important for the system to allow a transaction to abort without waiting for activity in the transaction's subtree to subside. In this case, a weak concurrent system might be an appropriate choice, even though the correctness conditions which they satisfy are weaker. Weak concurrent systems also appear to have further technical use, for example in providing simple explanations of the ideas used in "orphan detection" algorithms [11].

6.1. The weak concurrent controller

In this subsection, we define the weak concurrent controller. As we have already said, it is identical to the concurrent controller except that it has a more permissive ABORT operation. For convenience, we describe the controller here in its entirety. It has the same operations as the concurrent controller:

Input operations:

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

Output operations:

CREATE(T), T a non-access transaction,
INTERNAL_CREATE(T), T an access transaction,
COMMIT(T, v), ABORT(T),
INFORM_COMMIT_AT(X)OF(T), INFORM_ABORT_AT(X)OF(T).

Each state s of the concurrent controller consists of five sets: create_requested(s), created(s), commit_requested(s), commited(s), and aborted(s). The set commit_requested(s) is a set of (transaction, value) pairs, and the others are sets of transactions. (As before, we will occasionally write $T \in$ commit_requested(s) for $(T, v) \in$ commit_requested(s) for some v.) All are empty initially except for create_requested, which is $\{T_0\}$. Define returned(s) = committed(s) $\cup$ aborted(s). The operations are as follows.

- REQUEST_CREATE(T)
  Postcondition:

  create_requested(s) = create_requested(s') $\cup \{T\}$.

- REQUEST_COMMIT(T, v)
  Postcondition:

  commit_requested(s) = commit_requested(s') $\cup \{(T, v)\}$.

- CREATE(T), T a non-access transaction
  Precondition:

  $T \in$ create_requested(s') $-$ created(s');

  Postcondition:

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

- INTERNAL_CREATE(T), T an access transaction
  Precondition:

  $T \in$ create_requested(s') $-$ created(s');

  Postcondition:

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

- COMMIT(T, v)
  Precondition:

  $(T, v) \in$ commit_requested(s'), $T \notin$ returned(s'),
  children(T) $\cap$ create_requested(s') $\subseteq$ returned(s');

Postcondition:

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

- ABORT(T)
  Precondition:

  $$T \in \text{create-requested}(s') - \text{returned}(s');$$

  Postcondition:

  $$\text{aborted}(s) = \text{aborted}(s') \cup \{T\}.$$

- INFORM_COMMIT_AT(X)OF(T):
  Precondition:

  $$T \in \text{committed}(s').$$

- INFORM_ABORT_AT(X)OF(T):
  Precondition:

  $$T \in \text{aborted}(s').$$

Thus, the weak concurrent controller is permitted to abort any transaction that has had its creation requested, and which has not yet returned.

Lemma 53. *Let α be a schedule of the concurrent scheduler, and let s be a state which can result from applying α to the initial state. Then the following conditions are true.*

(1) *T is in* create_requested(s) *exactly if $T = T_0$ or α contains a* REQUEST_CREATE(T) *operation.*

(2) *If T is a non-access transaction, then T is in* created(s) *exactly if α contains a* CREATE(T)) *operation.*

(3) *If T is an access transaction, then T is in* created(s) *exactly if α contains an* INTERNAL_CREATE(T) *operation.*

(4) *(T, v) is in* commit_requested(s) *exactly if α contains a* COMMIT_REQUEST(T, v) *operation.*

(5) *(T, v) is in* committed(s) *exactly if α contains a* COMMIT(T, v) *operation.*

(6) *T is in* aborted(s) *exactly if α contains an* ABORT(T) *operation.*

6.2. Weak concurrent systems

The composition of transactions, resilient objects and the weak concurrent scheduler (lock managers and weak concurrent controller) is the *weak concurrent system.* A schedule of the weak concurrent system is a *weak concurrent schedule.*

Weak concurrent systems exhibit nice behavior to transactions except possibly to those which are descendants of aborted transactions. Thus, we say that a transaction T is an *orphan* in any sequence α of operations provided that an ancestor of T is aborted in α. In many of the properties we prove for weak concurrent systems, we will have to specify that the transactions involved are not orphans.

6.3. Properties of weak concurrent systems

As we did for serial and concurrent schedules, we here prove a number of useful basic properties for weak concurrent schedules. As before, most of these properties are simple to state and prove.

6.3.1. Operations in weak concurrent schedules

As before, we include a collection of lemmas describing the possible kinds and orders of operations that can occur in well-formed weak concurrent schedules. These lemmas are analogous to some in Section 5, and have similar proofs; the main difference is that we must take proper care with orphans. As before, we go on to show that all weak concurrent schedules are well-formed, so these properties actually follow just from the fact that these schedules are weak concurrent.

Lemma 54. *Let α be a well-formed weak concurrent schedule, and let $T \neq T_0$ be a transaction.*

(1) *If α contains any operation with transaction T, then α contains a* CREATE(T), *and a* REQUEST_CREATE(T).

(2) *If α contains a* COMMIT *for T, then α contains a* REQUEST_COMMIT *for T, a* CREATE(T) *and a* REQUEST_CREATE(T).

(3) *If α contains an* ABORT(T), *then α contains a* REQUEST_CREATE(T).

Lemma 55. *Let α be a well-formed weak concurrent schedule, and T a transaction. Assume that some descendant of T is in* transaction(α). *Then the following hold.*

(1) CREATE(T) *occurs in α.*

(2) *If $T \neq T_0$, then* REQUEST_CREATE(T) *occurs in α.*

Lemma 56. *Let α be a well-formed weak concurrent schedule, and let $T \neq T_0$.*

(1) *If α contains a* REQUEST_CREATE(T), *but does not contain a return operation for T, then* parent(T) *is not committed in α.*

(2) *If T is live in α, then* parent(T) *is not committed in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T), *or* ABORT(T), *then* parent(T) *is not committed in α.*

Proof. (1): Suppose a COMMIT operation for parent(T) occurs in α. Then the weak concurrent controller preconditions for the COMMIT operation imply that the COMMIT for parent(T) must be preceded by a REQUEST_COMMIT for parent(T). By well-formedness, the REQUEST_COMMIT for parent(T) must follow the REQUEST_CREATE(T), so that the COMMIT for parent(T) must follow the REQUEST_CREATE(T). Then the weak concurrent controller preconditions for the COMMIT operation imply that there must be a COMMIT operation for T in α, a contradiction.

(2) and (3) are as in paragraph 3.6.2. □

Lemma 57. *Let α be well-formed weak concurrent schedule, and let T be a transaction which is not an orphan in α.*

(1) *If α contains a* REQUEST_CREATE(T), *but does not contain a* COMMIT *operation for T, then all proper ancestors of T are live in α.*

(2) *If T is live in α, then all proper ancestors of T are live in α.*

(3) *If α contains a* REQUEST_CREATE(T) *but does not contain a* CREATE(T), *then all proper ancestors of T are live in α.*

Proof. By repeated use of the previous lemma, well-formedness and the weak concurrent controller preconditions. □

Lemma 58. *Let α be a well-formed weak concurrent schedule, and let T and T' be transactions with T' a descendant of T. Assume that T' is not an orphan in α and that there is a* COMMIT *operation for T in α.*

(1) *If there is a* REQUEST_CREATE(T') *in α, then there is a* COMMIT *operation for T' in α.*

(2) *If T' is in* transaction(α), *then there is a* COMMIT *operation for T' in α.*

Proof. (1): By Lemma 57.

(2): By Lemma 54 and part (1). □

6.3.2. Objects and locking

In this paragraph, we give two simple lemmas about the behavior of the locking strategy.

Lemma 59. *Let α be a weak concurrent schedule. Let X be an object, and let T and T' be accesses to X. Let U be an ancestor of T which is not an ancestor of T'. Assume that* CREATE(T) *precedes* CREATE(T') *in α.*

(1) *There is either an* INFORM_COMMIT_AT(X)OF(U), *or else an* INFORM_ABORT_AT(X) *for some ancestor of T, occurring between* CREATE(T) *and* CREATE(T') *in α.*

(2) *Either* CREATE(T') *is preceded by a* COMMIT *operation for U, and by a* REQUEST_COMMIT *operation for U, or else* CREATE(T') *is preceded by an* ABORT *operation for some ancestor of T.*

Proof. (1): By Lemma 44.

(2): By part (1) and the preconditions of the weak concurrent controller. □

Lemma 60. *Let α be a well-formed weak concurrent schedule, and X a basic object. Then the set of active transactions after $\alpha|R(X)$ is exactly the set of lockholders in the lock manager for X after α.*

Proof. By induction on the length of α. □

6.3.3. Well-formedness

Here, we show that every weak concurrent schedule is well-formed. It follows that all the properties proved earlier in this section are actually true for all weak concurrent schedules. From now on, we will use these properties without explicitly mentioning well-formedness.

Lemma 61. *Let α be a weak concurrent schedule. Then α is well-formed.*

Proof. By induction on the length of schedules. The base, length = 0, is trivial. Suppose that $\alpha\pi$ is a weak concurrent schedule, where π is a single operation, and assume that α is well-formed. If π is an output of a primitive P, then the result is immediate since each primitive preserves well-formedness. No INTERNAL_CREATE operation can cause a violation. So assume that π is an input to a primitive P. It suffices to show that $\alpha\pi|P$ is well-formed. There are six cases.

(1) π is CREATE(T) and T is a non-access transaction. The controller preconditions ensure that CREATE(T) does not appear in α.

(2) π is CREATE(T) and T is an access to resilient object $R(X)$. By the lock manager preconditions, no CREATE(T) appears in α. The lock manager preconditions and Lemma 60 imply that all the transactions which are active after α are ancestors of T.

(3) π is COMMIT(T, v). Then π is an input to transaction parent(T). Weak concurrent controller preconditions imply that α contains REQUEST_COMMIT(T, v), and so Lemma 54 implies that α contains REQUEST_CREATE(T). Also, weak concurrent controller preconditions ensure that α does not contain a return operation for T.

(4) π is ABORT(T). Then π is an input to transaction parent(T). Weak concurrent controller preconditions imply that α contains a REQUEST_CREATE(T). Weak concurrent controller preconditions ensure that α does not contain a return operation for T.

(5) π is INFORM_COMMIT_AT(X)OF(T) at resilient object $R(X)$. By the preconditions of the weak controller, α contains a COMMIT for T. If INFORM_ABORT_AT(X)OF(T) occurs in α, then α also contains an ABORT for T, which contradicts weak concurrent controller preconditions. Thus, no INFORM_ABORT_AT(X)OF(T) occurs in α. Since a COMMIT for T occurs in α, weak concurrent controller preconditions imply that a REQUEST_COMMIT for T also occurs in α.

(6) π is INFORM_ABORT_AT(X)OF(T) at resilient object $R(X)$. By the preconditions of the weak concurrent controller, α contains ABORT(T). If INFORM_COMMIT_AT(x)OF(T) occurs in α, then α contains a COMMIT for T, which contradicts weak concurrent controller preconditions. Thus, no INFORM_COMMIT_AT(X)OF(T) occurs in α. □

6.3.4. Visibility and weak concurrent schedules

This paragraph states and proves important properties involving visibility in weak concurrent schedules. In particular, the most important result of this paragraph is Lemma 66, which relates the portion of a weak concurrent schedule which is visible to a particular transaction, to schedules of transactions and basic objects. The first lemma shows how visibility propagates among the transactions during a weak concurrent execution.

Lemma 62. *Let $\alpha\pi$ be a weak concurrent schedule, where π is a single operation.*

(1) *If π is* CREATE(T), *then* visible($\alpha\pi$, T) = visible(α, parent(T))π.

(2) *If π is* COMMIT(T, v), *then* visible($\alpha\pi$, parent(T)) = (visible(α, T))π.

(3) *If π is* ABORT(T), *then* visible($\alpha\pi$, parent(T)) = visible(α, parent(T))π.

(4) *If π is* COMMIT(T, v), *and T' is a descendant of* parent(T) *but not T, then* visible($\alpha\pi$, T') − visible($\alpha\pi$, parent(T)) = visible(α, T') − visible(α, T).

Proof. (1): By Lemma 55, π is the first serial operation in $\alpha\pi$ whose transaction is a descendant of T, and T is not visible to parent(T). Thus, any transaction other than T visible to T in $\alpha\pi$ is visible to parent(T) in α. Then parent(T) is visible to T in $\alpha\pi$, and by Lemma 8, visible($\alpha\pi$, parent(T))π = visible($\alpha\pi$, T).

By the definition of visibilty, any transaction visible to parent(T) in $\alpha\pi$ is visible to parent(T) in α, and visible(α, parent(T)) = visible($\alpha\pi$, parent(T)). Substituting into the equality above, we have the result.

(2): By the definition of visibility, any transaction visible to parent(T) in $\alpha\pi$ is either visible to parent(T) in α, or is visible to T in α. But any transaction visible to parent(T) in α is visible to T in α, so we have that any transaction visible to parent(T) in $\alpha\pi$ is visible to T in α, and visible($\alpha\pi$, parent(T)) is a subsequence of visible(α, T)π. It follows immediately from the definition of visibility that any transaction visible to T in α is visible to parent(T) in $\alpha\pi$, so that visible(α, T) is a subsequence of visible($\alpha\pi$, parent(T)). The result is immediate.

(3): Immediate from the definition of visibility.

(4): Clearly, visible(α, T') is a subsequence of visible($\alpha\pi$, T'). Any operation in visible($\alpha\pi$, T') − visible(α, T') has a transaction which is a descendant of T, and so is either π or is visible to T in α, and thus is in visible(α, T)π. Thus we have

$$\text{visible}(\alpha\pi, T') - \text{visible}(\alpha, T)\pi = \text{visible}(\alpha, T') - \text{visible}(\alpha, T)\pi.$$

As π is not in visible(α, T'), this equals visible(α, T') − visible(α, T). By part (2), visible($\alpha\pi$, parent(T)) = visible(α, T)π, and the result follows by substitution in the first identity. □

Now we prove two lemmas involving visibility and the behavior of resilient objects in weak concurrent systems.

Lemma 63. *Let α be a weak concurrent schedule. Let $R(X)$ be a resilient object, and let T and T' be accesses to $R(X)$. If T' is live and not an orphan in α and* CREATE(T)

occurs in α, then either T is visible to T' in α, or else CREATE(T) *is in the scope of an* INFORM_ABORT_AT(X)OF(U) *in $\alpha|R(x)$.*

Proof. There are two cases.

(1) CREATE(T) precedes CREATE(T') in α. Assume T is not visible to T' in α. Then Lemma 59 implies that there is an INFORM_ABORT_AT(X) operation for some ancestor of T, occurring after CREATE(T) in α.

(2) CREATE(T') precedes CREATE(T) in α. Then Lemma 59 implies that there is either a COMMIT or an ABORT for some ancestor of T', in α. Since T' is not an orphan in α, there is a COMMIT for an ancestor of T' in α. Then Lemma 58 implies that T' is returned in α, a contradiction. □

Lemma 64. *Let α be a weak concurrent schedule. Let $R(X)$ be a resilient object, let T and T' be accesses to $R(X)$, and let T'' be any transaction. Assume that T' is not an orphan in α. If an operation π of T precedes an operation π' of T' in α, π is not in the scope of an* INFORM_ABORT *and T' is visible to T'' in α, then T is visible to T'' in α.*

Proof. By well-formedness, CREATE(T) and CREATE(T') are operations in α, in that order. Let α' be the prefix of α ending with CREATE(T'). Then T' is live and not an orphan in α'. By Lemma 63, T is visible to T' in α', and so in α. Lemma 8 implies that T is visible to T'' in α. □

The following lemma is straightforward.

Lemma 65. *Let α be a weak concurrent schedule, and let T be a transaction which is not an orphan in α. Any transaction T' visible to T in α is not an orphan in α.*

Proof. If T' is an ancestor of T, the result is immediate. Otherwise, COMMIT operations appear in α for every proper descendant of lca(T, T') that is an ancestor of T'. By well-formedness, none of these transactions has aborted. Since the remaining ancestors of T' are also ancestors of T, the result follows. □

We are now ready to prove the key lemma of this paragraph.

Lemma 66. *Let α be a weak concurrent schedule, let T not be an orphan in α, and let P be a resilient primitive.*

(1) *If P is a transaction T', then* visible(α, T)$|T'$ *is a prefix of $\alpha|T'$ and a schedule of T'.*

(2) *If P is a resilient object $R(X)$, then* visible(α, T)$|R(X)$ *is a prefix of* undo($\alpha|R(X)$) *and a schedule of basic object X.*

Proof. (1): Immediate from Lemmas 11 and 1.

(2): First, we show that any operation in visible(α, T)|$R(X)$ also occurs in undo($\alpha|R(X)$). If π is in visible(α, T)|$R(X)$, it means that all ancestors of transaction(π) up to lca(transaction(π), T) have committed. By assumption, T is not an orphan in α, so Lemma 65 implies that transaction(π) is not an orphan in α. Thus, by the preconditions of the weak concurrent controller there is no INFORM_ABORT for any ancestor of transaction(π) in α. Therefore, π is in undo($\alpha|R(X)$).

Now we consider any two operations π and π' of undo($\alpha|R(X)$), where π precedes π'. Assume that π' is in visible(α, T)|$R(X)$. Let T'' = transaction(π) and T' = transaction(π'). Then T' is visible to T in α, and T' is not an orphan in α by Lemma 65. Since π is in undo($\alpha|R(X)$), no INFORM_ABORT occurs at $R(X)$ for any ancestor of T'' in α, with π in its scope. Then Lemma 64 implies that T'' is visible to T in α. Thus, π is in visible(α, T)|$R(X)$. It follows that visible(α, T)|$R(X)$ is a prefix of undo($\alpha|R(X)$).

By Lemma 61, $\alpha|R(X)$ is a well-formed schedule of resilient object $R(X)$. Then the resiliency condition implies that undo($\alpha|R(X)$) is a schedule of basic object X. So by Lemma 1, visible(α, T)|$R(X)$ is a schedule of basic object X. □

Finally, we prove that, in a weak concurrent schedule, concurrently executing transactions access disjoint sets of resilient objects.

Lemma 67. *Let α be a weak concurrent schedule, and let T and T' be transactions which are not orphans in α. Let T'' = lca(T, T'). Let β = visible(α, T) − visible(α, T'') and β' = visible(α, T') − visible(α, T''). Then no resilient object has operations in both β and β'.*

Proof. The result is trivial if T is an ancestor of T' or vice versa, so assume the contrary. Then T'' is neither T nor T'.

Let $R(X)$ be a resilient object such that both β and β' contain operations of $R(X)$. Thus, there are two accesses to X, U and V, such that operation π of U and operation ϕ of V occur in β and β' respectively. Then U is visible to T, V is visible to T', but neither U nor V is visible to T'', in α. It follows that U is not visible to T' and V is not visible to T in α. In particular, $U \neq V$.

By well-formedness, we can assume without loss of generality that π = CREATE(U) and ϕ = CREATE(V). We can also assume without loss of generality that π precedes ϕ in α. Since T and T' are not orphans in α, Lemma 65 implies that U and V are not orphans in α. Then Lemma 59 implies that U is visible to V in α. But then U is visible to T' in α, a contradiction. □

7. Simulation of serial systems by concurrent systems

In this section, we prove the main results of this paper: that concurrent schedules are serially correct, and that weak concurrent schedules are correct at T_0. Both these

results follow from an interesting theorem about weak concurrent schedules, which says that the portion of any weak concurrent schedule which is visible to a live non-orphan transaction is equivalent to (i.e., looks the same at *all* primitives as) a serial schedule.

The proof of this theorem is quite interesting, as it provides considerable insight into the scheduling algorithm. The proof shows not only that a transaction's view of a weak concurrent schedule is equivalent to *some* serial schedule, but by a recursive construction, it actually produces such a schedule. It is interesting and instructive to observe how the views that different transactions have the system execution get passed up and down the transaction tree, as CREATES, COMMITS and ABORTS occur.

Theorem 68. *Let α be a weak concurrent schedule, and T any transaction which is live and not an orphan in α. Then there is a serial schedule β which is equivalent to* visible(α, T).

Proof. We proceed by induction on the length of α. The basis, length = 0, is trivial. Fix α of length at least 1, and assume that the claim is true for all shorter weak concurrent schedules. Let π be the last operation of α, and let $\alpha = \alpha'\pi$. Fix T which is live and not an orphan in α. We must show that there is a serial schedule β which is equivalent to visible(α, T).

If π is not a serial operation, then

$$\begin{aligned}\text{visible}(\alpha', T) &= \text{visible}(\text{serial}(\alpha'), T) = \text{visible}(\text{serial}(\alpha), T)\\ &= \text{visible}(\alpha, T),\end{aligned}$$

and the result is immediate by induction. So we can assume that π is a serial operation. Also, if transaction(π) is not visible to T in α, then visible(α, T) = visible(α', T), and the results is again immediate by induction. Thus, we can assume that transaction(π) is visible to T in α. Also, T is not an orphan in α'. There are four cases.

(1) π is an output operation of a transaction or resilient object. Then the inductive hypothesis implies the existence of a serial schedule β' which is equivalent to visible(α', T). Let $\beta = \beta'\pi$. We must show that β is equivalent to visible(α, T) and serial.

Let P be any primitive. Then

$$\begin{aligned}\beta|P = \beta'\pi|P &= \text{visible}(\alpha', T)\pi|P && \text{by inductive hypothesis}\\ &= \text{visible}(\alpha, T)|P && \text{by Lemma 12.}\end{aligned}$$

Therefore, β is equivalent to visible(α, T).

Let π be an output of primitive P. Then $\beta|P = \text{visible}(\alpha, T)|P$ by equivalence, which is a schedule of P by Lemma 66. Lemma 4 implies that β is serial.

(2) π is a CREATE(T') operation. Then transaction(π) = T', and so T' is visible to T in α. Then Lemma 55 implies that π is the first operation whose transaction

is a descendant of T'. Then, by the definition of visibility, it must be that $T' = T$. By Lemma 57, parent(T) is live in α'. Since parent(T) is not an orphan, the inductive hypothesis implies the existence of a serial schedule β' which is equivalent to visible(α', parent(T)). Let $\beta = \beta'\pi$. We must show that β is equivalent to visible(α, T) and serial.

Let P be any primitive. Then

$$\begin{aligned}\beta|P &= \beta'\pi|P = \text{visible}(\alpha', \text{parent}(T))\pi|P && \text{by inductive hypothesis}\\ &= \text{visible}(\alpha, T)|P && \text{by Lemma 62.}\end{aligned}$$

Thus, β is equivalent to visible(α, T).

Consider any execution of the serial system having β' as its operation sequence, and let s' be the state of the serial scheduler after β'. We show that π is enabled in s'. That is, we show that $T \in \text{create\_requested}(s')$, that $T \notin \text{created}(s')$, that $T \notin \text{aborted}(s')$, and that $\text{siblings}(T) \cap \text{created}(s') \subseteq \text{returned}(s')$.

Consider any execution of the weak concurrent system having α as its operation sequence, and let s be the state of the weak concurrent scheduler after α'. State s contains a component s_c for the weak concurrent controller and a component s_X for the lock manager for each object X.

If $T = T_0$, then $T \in \text{create\_requested}(s')$ by the initial conditions. If $T \neq T_0$, then $T \in \text{create\_requested}(s_c)$ by the preconditions of the concurrent scheduler, so a REQUEST_CREATE(T) operation occurs in α'. The REQUEST_CREATE(T) operation has transaction parent(T), and so is in visible(α', parent(T)), and thus is in β'. Therefore, $T \in \text{create\_requested}(s')$.

If $T \in \text{created}(s')$, then there is a CREATE(T) operation in β', and hence in α'. Then α would have two such operations, which is impossible, so $T \notin \text{created}(s')$.

If $T \in \text{aborted}(s')$, there is an ABORT(T) operation in β', and hence in α'. Then α would have an ABORT(T) followed by a CREATE(T). This is impossible, so $T \notin \text{aborted}(s')$.

Assume $U \in \text{siblings}(T) \cap \text{created}(s')$. Then there is a CREATE(U) operation in β', and so in visible(α', parent(T)). Since CREATE(U) occurs at U, U is visible to parent(T) = parent(U) in α'; thus, COMMIT(U, u) occurs in α', for some u. Since COMMIT(U, u) occurs at parent(T), COMMIT(U, u) is in visible(α', parent(T)), and so in β'. Thus, $U \in \text{returned}(s')$.

(3) π is a COMMIT(T', v) operation. Then $T'' = \text{parent}(T') = \text{transaction}(\pi)$ is visible to T and not an orphan in α. Also, T' is not an orphan in α', by Lemma 65. Then, since α is well-formed, T' is live in α' and so, by Lemma 57, T'' is live in α' and so in α. Since T'' is live and visible to T in α, T'' is an ancestor of T. Since T is live in α, Lemma 58 implies that T is not a descendant of T'. The inductive hypothesis yields two serial schedules, β' and β'', which are equivalent to visible(α', T') and visible(α', T) respectively. Let $\gamma = \text{visible}(\beta', T'')$. Let $\beta_1 = \beta' - \gamma$ and $\beta_2 = \beta'' - \gamma$. We show that $\beta = \gamma\beta_1\pi\beta_2$ is equivalent to visible(α, T) and serial.

Lemma 28 implies that γ is a serial schedule.

Since T'' is visible to T' in α', Lemma 10 implies that

$$\text{visible}(\alpha', T'') = \text{visible}(\text{visible}(\alpha', T'), T''),$$

which is equivalent to $\text{visible}(\beta', T'') = \gamma$; thus γ is equivalent to $\text{visible}(\alpha', T'')$. Also, since T'' is visible to T in α', Lemma 10 implies that

$$\text{visible}(\alpha', T'') = \text{visible}(\text{visible}(\alpha', T), T''),$$

which is equivalent to $\text{visible}(\beta'', T'')$. Thus, γ is also equivalent to $\text{visible}(\beta'', T'')$.

Then, by Lemma 31 (applied with $\text{serial}(\alpha')$ as the schedule α hypothesized in the lemma), $\gamma\beta_1$ and $\gamma\beta_2$ are serial schedules which are equivalent to β' and β'', respectively.

We have that $\text{visible}(\alpha, T'') = \text{visible}(\alpha', T')\pi$ by Lemma 62, which is equivalent to $\beta'\pi$, which is in turn equivalent to $\gamma\beta_1\pi$. That is, $\text{visible}(\alpha, T'')$ is equivalent to $\gamma\beta_1\pi$.

Since β'' is equivalent to $\text{visible}(\alpha', T)$ and γ is equivalent to $\text{visible}(\alpha', T'')$, by Lemma 10, $\beta_2 = \beta'' - \gamma$ is equivalent to

$$\text{visible}(\alpha', T) - \text{visible}(\alpha', T'') = \text{visible}(\alpha, T) - \text{visible}(\alpha, T'')$$

by Lemma 62.

Thus, β is equivalent to $\text{visible}(\alpha, T'')(\text{visible}(\alpha, T) - \text{visible}(\alpha, T''))$. Since T'' is visible to T in α, by Lemma 8, it is easy to see that the same operations appear in this schedule as in $\text{visible}(\alpha, T)$. Let P be any primitive. Then $\text{visible}(\alpha, T'')|P$ is a prefix of $\text{visible}(\alpha, T)|P$, by Lemma 66. It follows that $\beta|P = \text{visible}(\alpha, T)|P$, so that β is equivalent to $\text{visible}(\alpha, T)$.

It remains to show that β is serial. This follows from Lemma 32, provided we can show that:

(3a) $\gamma\beta_1\pi$ is a serial schedule,

(3b) T' sees everything in $\gamma\beta_1$,

(3c) T sees everything in $\gamma\beta_2$,

(3d) $\gamma = \text{visible}(\gamma\beta_1, T'') = \text{visible}(\gamma\beta_2, T'')$ and

(3e) no basic object has operations in both β_1 and β_2.

(3a): Consider any execution of the serial system having $\gamma\beta_1$ as its operation sequence, and let s' be a state of the serial scheduler after $\gamma\beta_1$. We show that π is enabled in state s'. That is, we show that $(T', v) \in \text{commit\_requested}(s')$, that $T' \notin \text{returned}(s')$, and that $\text{children}(T') \cap \text{create\_requested}(s') \subseteq \text{returned}(s')$.

Consider any execution of the weak concurrent system having α as its operation sequence, and let s be the state of the weak concurrent scheduler after α' with components s_c (the weak controller state) and s_X for every object X (the lock managers).

Since the weak concurrent scheduler is able to perform COMMIT(T', v) in state s, it must be that (T', v) is in $\text{commit\_requested}(s_c)$, and so it must be that T' issues a REQUEST_COMMIT(T', v) in α'. Since T' is visible to itself, and β' is equivalent

to visible(α', T'), it follows that this REQUEST_COMMIT(T', v) operation also occurs in $\gamma\beta_1$. Therefore, (T', v) is in commit_requested(s').

Since α is well-formed, at most one return operation for T' appears in α; thus, T' is not in returned(s').

Fix $U \in$ children(T') $\cap$ create_requested(s'). Then REQUEST_CREATE(U) is performed at T' in $\gamma\beta_1$, and hence in α', so $U \in$ create_requested(s_c). Since the weak concurrent scheduler is able to perform COMMIT(T', v) in state s, it must be that $U \in$ returned(s_c). Therefore, a return operation for U is performed at T' in α'. Since T' is visible to itself and $\gamma\beta_1$ is equivalent to visible(α', T'), this implies that the return for U also occurs at T' in $\gamma\beta_1$. Therefore, U is in returned(s').

(3b): Immediate from Lemma 10.

(3c): Immediate from Lemma 10.

(3d): We have that γ is equivalent to both visible(β', T'') and visible(β'', T''), and that $\gamma\beta_1$ and $\gamma\beta_2$ are equivalent to β' and β'' respectively. By Lemma 10, γ is equivalent to both visible($\gamma\beta_1$, T'') and visible($\gamma\beta_2$, T''). Equality follows.

(3e): Immediate from Lemma 67.

(4) π is an ABORT(T') operation. Then $T'' =$ parent(T') = transaction(π) is visible to T in α, and so is not an orphan in α, by Lemma 65. Then T' is live in α' and, by Lemma 57, T'' is live in α' and so in α. Since T'' is live and visible to T in α, T is a descendant of T''. Since T is not an orphan in α, T is not a descendant of T'. The inductive hypothesis yields two serial schedules, β' and β'', which are equivalent to visible(α', T'') and visible(α', T) respectively. Let $\beta_1 = \beta'' - \beta'$. We show that $\beta = \beta'\pi\beta_1$ is equivalent to visible(α, T) and serial.

By Lemma 31, $\beta'\beta_1$ is a serial schedule which is equivalent to β''.

Let P be a primitive other than T''. Then

$$\beta|P = \beta'\beta_1|P = \beta''|P = \text{visible}(\alpha', T)|P = \text{visible}(\alpha, T)|P$$

by Lemma 62. Also, since T'' is visible to T in α,

$$\begin{aligned}\text{visible}(\alpha, T)|T'' &= \text{visible}(\alpha, T'')|T'' = \text{visible}(\alpha', T'')\pi|T'' \quad \text{by Lemma 62}\\ &= \beta'\pi|T'' = \beta|T''.\end{aligned}$$

Thus β is equivalent to visible(α, T).

It remains to show that β is serial. This follows from Lemma 33, provided we can show that

(4a) $\beta'\pi$ is a serial schedule,

(4b) T sees everything in $\beta'\beta_1$, and

(4c) $\beta' = \text{visible}(\beta', T'') = \text{visible}(\beta'\beta_1, T'')$.

(4a): Consider any execution of the serial system having β' as its operation sequence, and let s' be a state of the serial scheduler after β'. We show that π is enabled in state s'. That is, we show that $T' \in$ create_requested(s'), that $T' \notin$ created(s'), $T' \notin$ aborted(s'), and that siblings(T') $\cap$ created(s') $\subseteq$ returned(s').

Consider any execution of the weak concurrent system having α as its operation sequence, and let s be the state of the weak concurrent scheduler after α' with

components s_c (the weak controller state), and s_X for every object X (the lock managers).

Since the weak concurrent scheduler is able to perform ABORT(T') in state s, it must be that T' is in create_requested(s_c), and so it must be that T'' issues a REQUEST_CREATE(T') in α'. Since T'' is visible to itself, and β' is equivalent to visible(α', T'), it follows that this REQUEST_CREATE(T') operation also occurs in β'. Therefore, T' is in create_requested(s').

Since α cannot contain two ABORT(T') operations, there cannot be an ABORT(T') operation in α', and so there cannot be one in β'. Assume that there is a CREATE(T') in β'. Then T' is visible to T'' in α', so COMMIT(T') occurs in α'. But then a COMMIT(T') and ABORT(T') both occur in α, which cannot occur. Therefore, there is neither an ABORT(T') nor a CREATE(T') in β', and so T' is neither in aborted(s') not in created(s').

Fix $U \in$ siblings(T') $\cap$ created(s'). Then there is a CREATE(U) in β'. But then U is visible to T'' in α', so that a COMMIT for U occurs in α', and hence (because parent(U) is visible to T'' in α') a COMMIT for U occurs in β'. Therefore, $U \in$ returned(s').

(4b): Immediate from Lemma 10.

(4c): The first equality follows from Lemma 10. Clearly, $\beta' =$ visible(β', T'') is a prefix of visible($\beta'\beta_1$, T''). Equality follows because any operation in β_1 visible to T'' in $\beta'\beta_1$ would also be visible to T'' in α', and so would be in β' and not β_1. □

Corollary 69. *Every weak concurrent schedule is serially correct for every non-orphan non-access transaction.*

Proof. Let α be a weak concurrent schedule. Let T be a non-access transaction that is not an orphan in α. We must show that $\alpha|T$ is a serial schedule. Note that T is not an orphan in any prefix of α. There are three cases:

(1) $\alpha|T$ is empty. Then the result is trivial.

(2) T is live in α. Then Theorem 68 yields a serial schedule β that is equivalent to visible(α, T). Thus, $\alpha|T =$ visible(α, T)$|T = \beta|T$, which suffices.

(3) T is a transaction which is live in some proper prefix of α, but is not live in α. Since α is well-formed, α has a prefix $\alpha'\pi$, where π is a COMMIT operation for T, $\alpha'|T = \alpha|T$ and T is live in α'. Then Theorem 68 yields a serial schedule β that is equivalent to visible(α', T)$|T$. Thus,

$$\alpha|T = \alpha'|T = \text{visible}(\alpha', T)|T = \beta|T,$$

which suffices. □

Now, since T_0 cannot become an orphan (having no ancestors to abort), our first major correctness result is immediate.

Corollary 70. *Every weak concurrent schedule is serially correct for T_0.*

Having proved correctness properties for weak concurrent schedules, we are now ready to prove the correctness of concurrent schedules.

Lemma 71. *Every concurrent execution is a weak concurrent execution.*

Proof. The proof is by induction on execution length, with a trivial basis. Let $\gamma = \gamma', s', \pi, s$ be a concurrent execution with (s', π, s) a single step of the concurrent system, and assume the lemma holds for γ'. Let s'_c and s_c denote the states of the concurrent controller in system states s' and s.

If π is any operation other than an ABORT, the result is immediate since the postconditions for all other operations are identical in the concurrent and weak concurrent systems, and the preconditions are either identical or are stronger in the concurrent system than in the weak concurrent system. Assume that π is an ABORT(T). We must show that $T \in \text{create\_requested}(s'_c) - \text{returned}(s'_c)$.

Since π is enabled in state s'_c in the concurrent controller,

$$T \in (\text{create\_requested}(s'_c) - \text{created}(s'_c) - \text{aborted}(s')) \cup (\text{commit\_requested}(s'_c) - \text{returned}(s'_c)).$$

If T is in $\text{create\_requested}(s'_c) - \text{created}(s'_c) - \text{aborted}(s'_c)$, Lemma 45 implies that γ' contains no CREATE(T) or ABORT(T) operation. By well-formedness, γ' also contains no COMMIT operation for T, and the result follows from Lemma 45. On the other hand, if T is in $\text{commit\_requested}(s'_c) - \text{returned}(s'_c)$, Lemma 45 implies that a REQUEST_COMMIT operation for T occurs in γ'. By well-formedness, this is preceded by a CREATE(T) operation and, by the concurrent controller precondition, this is preceded by a REQUEST_CREATE for T. Finally, again by Lemma 45, the result follows. □

Now we can prove the second major result of the paper.

Corollary 72. *Every concurrent schedule is serially correct.*

Proof. Let α be a concurrent schedule. Then α is also a weak concurrent schedule, by Lemma 71, and is well-formed, by Lemma 61. We must show that α is serially correct for every transaction T. There are three cases.

(1) $\alpha|T$ is empty. Then the result is trivial.

(2) T is live in α. By Lemma 50, all of T's ancestors are live in α, so that T is not an orphan in α. Then Corollary 69 yields the result.

(3) T is a transaction which is live in some proper prefix of α, but is not live in α. By Lemma 51, α has a prefix $\alpha'\pi$, where π is a return operation for T, $\alpha'|T = \alpha|T$

and T is live in α'. By Lemma 50, all of T's ancestors are live in α', so T is not an orphan in α'. Then Corollary 69 implies that α' is serially correct for T, so that α is serially correct for T. □

For completeness, we include an analogue of Theorem 68 for concurrent schedules.

Theorem 73. *Let α be a concurrent schedule, and T any transaction which is live in α. Then there is a serial schedule β which is equivalent to* visible(α, T).

Proof. Lemma 71 implies that α is a weak concurrent schedule. Since T is live in α, Lemma 50 implies that T is not an orphan in α. Then Theorem 68 yields the result. □

8. Discussion

In this paper, we have presented a formal model for describing nested transaction systems and their properties. The model has many features that we believe make it a major contribution to the understanding of transaction systems, and we highlight some of these below.

First, the entire model is based on a very general and very simple underlying model for concurrent computation, the I/O automaton model. The general definitions and properties of this underlying model provide the necessary underpinnings for our entire transaction modelling effort. This modelling is very easy to learn and use, and its usefulness extends much beyond transaction systems. Thus, it seems to us to be a very satisfactory foundation for our work.

Our transaction system model permits simple, yet completely rigorous description of concurrency control algorithms in ways which correspond very closely to the usual informal ways of understanding the algorithms. The important components of transaction systems, the transactions, data and schedulers, are described explicitly, which greatly facilitates reasoning about them.

There is a substantial amount of work in this area which does not represent all of these components explicitly, but only implicitly, by properties of their behavior [3, 9, 19, for example]. There are problems with this approach. A key ingredient that is usually absent from such implicit models is a clear notion of "causality", describing how particular actions (operations) are triggered by other actions or states. In contrast, our explicit representation of all system components as I/O automata makes it easy to understand exactly what causes all operations to occur. When causality is important in reasoning about algorithms, as in [9], implicit models can be extraordinarily difficult to use. Even in cases where implicit models can be used, we see the present work as providing a formal and intuitive basis for that work.

Our model represents transactions as essentially arbitrary automata, subject only to simple syntactic constraints. This approach is much more general than representing them as programs in some particular, overly-constrained language.

The I/O automata model permits description of algorithms in an abstract form which is not tied to a particular programming language or system, and which allows maximum nondeterminism. An implementation of an algorithm for a particular system will generally restrict the nondeterminism allowed in our presentation, because of the need to tailor the implementation to the requirements of a particular environment. However, since the implementation is just a restriction of the abstract algorithm, correctness properties of the algorithm within our model will hold a fortiori for the implementation.

Formulating nested transaction systems as I/O automata permits precise formulation of the correctness conditions to be satisfied by concurrency control algorithms; those correctness conditions can be stated at the transaction interface, an interface which does not contain explicit information about object representation. Because of this choice of interface, the correctness conditions can be stated in a robust way: the same conditions can be useful for describing the properties of many different kinds of algorithms, some of which manipulate the data in very different ways. Also, the correctness conditions can be described in a way that is meaningful to a user of the system.

The particular correctness conditions that we describe in this paper involve serial correctness at transaction interfaces. We believe that these particular correctness definitions are very interesting, and will be useful for describing the correctness of most of the usual algorithms studied in the concurrency control area. That is, the same conditions appear to be the right ones to use to describe correctness of many different kinds of algorithms, including those that use locking, timestamps, multiple versions, and replicated data.

The model permits rigorous correctness proofs to be carried out for concurrency control algorithms in ways that follow intuitive understanding of the algorithms. For example, in this paper, we have used the model to describe and show the correctness of a very important nested transaction concurrency control algorithm. Our correctness proofs are constructive and provide considerable intuition about the workings of the algorithm. In contrast to most correctness proofs for concurrent algorithms, our proofs are not voluminous low-level case-analyses; rather, they consist of a large number of clear and natural lemmas about the behavior of the algorithm. These lemmas can be understood individually, and build upon each other in the manner of good mathematics. Many of the lemmas should be reusable in extensions of this work as well.

A successful model of nested transactions should contain the classical theory as a special case, in a way which is natural and sheds some light on that case. We believe that our model has contributed much to the classical theory. For example, the I/O automaton model provides a general underlying model, a missing component of the classical theory. Also, our explicit and general modelling of the transactions

unifies the earlier collection of somewhat arbitrary approaches. Our use of the transaction interface for stating correctness conditions is also an improvement.

Another contribution to the classical theory is in motivating serializability as a correctness condition. Serializability consists of two criteria: individually, each transaction must see a consistent state, and together, they must appear to run in a serial order. (A schedule in which each transaction reads and writes the initial state of the database provides a consistent state to each transaction, but is not serializable.) Why is this second ordering property a part of the generally accepted correctness condition of the classical theory? Clearly, because of implicit nesting in the context of the transaction system. In practice, transactions perform tasks on behalf of some external entity or entities, which may expect one transaction to see the results of the next. In the natural formulation of classical systems within the present model, the classical transactions are children of T_0, with data accesses as their only children. The root is an explicit representation of the external environment in which the system runs. Thus, the ordering property of serializability is a natural consequence of the requirement that all transactions see serial schedules, including T_0. It does not have to be introduced as an independent requirement in need of separate justification.

By now, there has been a large amount of systems design and algorithms work that uses or implements nested transactions. It seems likely that these ideas will form the basis of future programming languages for distributed computing. However, there is currently a problem with the presentation of this work. Some of these algorithms are presented in the context of specific systems and programming languages. Very useful and general ideas are too intimately connected with details of the systems to be fully appreciated, particularly for readers with only a passing understanding of those systems. This level of detail also makes careful reasoning about the algorithms very difficult.

We believe that our model has provided the necessary framework and some of the necessary vocabulary for describing this work in a clear and unambiguous way. We are currently studying much of this work on systems design and algorithms using our model, and our preliminary results indicate that it works very well.

Throughout the paper, we have described connections with other people's work as appropriate. Here, we mention some of the particular modelling work that relates most closely to ours, and describe the connections in more detail.

First, the pioneering work of Bernstein and Goodman [5, etc.] has had a strong influence on this work. Quite early, they recognized the need for a model for single-level transaction systems, that would have many of the characteristics which we have sought for nested transaction systems. They have carried out extensive research on precise understanding of single-level transaction concurrency control algorithms. They have presented formal statements of correctness conditions, in terms of serializability of the accesses to data objects by different transactions. They have described some concurrency control algorithms with precision, and have proved correctness of some algorithms, using a lemma which characterizes serializability

by absence of cycles in a certain dependency relation. Their work has gone a long way toward providing precise understanding of the work in this area.

However, the particular models used by Bernstein and Goodman have some problems which limit their applicability. For instance, the basic correctness condition is stated in terms of the interface between the data objects and the algorithm. There are many algorithms which handle objects in very different ways, e.g., using multiple versions, or making multiple copies in order to permit "backing out" of operations. Since these algorithms do not preserve the specified object interface, they would not be considered correct under the same correctness condition. Thus, the correctness condition must be modified. Another limitation is that the proof technique, which involves proving absence of cycles, is a proof by contradiction; it does not give much insight into the operation of the algorithms. For many reasons, it is not at all clear how to extend these frameworks to handle nesting of transactions.

Earlier attempts in [4, 9, 19] to model nested transactions have made significant contributions. For example, [19] contains a language-independent model, which is used to give precise correctness conditions and a proof for a locking algorithm. Many of the ideas in that work have been useful in providing a vocabulary for talking about nested transactions. However, attempts to extend the model of [19] to handle correctness of orphans, as [9] demonstrates, are not sufficiently expressive. Certain aspects of the model lead to technical difficulties; for example, it fails to model the transactions explicitly, using instead a specification of their behavior. Our new model builds on the strengths of the earlier work, while managing (we believe) to avoid its weaknesses.

Finally, the very recent work in [3] proposes another general model for nested transactions. While on the surface the models appear quite different, they are actually "compatible", in that the concepts described in [3] seem to be easily definable within our model. The style of the model in [3] is different from ours: their work models transactions and the scheduler implicitly, for instance. However, we believe that their important axiomatic statements of properties can be described as assumptions and lemmas about behaviors of components in our model. Also, the partial orders which they use to model executions can actually be defined simply and directly in terms of our linearly-ordered executions. There are many points of agreement: the use of the transaction interface for stating correctness conditions, and the use of the virtual root transaction T_0, to mention two.

On the other hand, the emphasis in [3] is on a different example from the one studied in this paper. They consider multiple levels of abstraction for the data, and regard transactions at any level of the transaction tree as accesses to data at a corresponding level of abstraction. This view meshes quite well with our model, where, for example, we use the same CREATE notation for creation of a transaction and invocation of an operation on data. Their paper clarifies the concurrency control requirements for data at different levels, when the correctness condition is serial correctness at T_0. We hope and expect that it will be easy to restate their results as claims about our model.

We note that the work in [3] only treats concurrency control, but does not address the very critical and difficult issues of resiliency.

9. Further work

This paper is an embarkation on a major project to formulate a unified presentation of the most important algorithms for concurrency control and resiliency, especially those for nested transactions. In this paper, we have defined a general framework meeting the requirements outlined above. We have demonstrated the power of this framework by using it to specify two correctness conditions for nested transactions, to present two locking algorithms for implementing nested transactions, and to prove that the algorithms satisfy their respective requirements.

It is possible that additional work might yield shorter and more elegant correctness proofs for the algorithms of this paper. One idea which might be helpful is suggested by the close relationship between each resilient object and its corresponding lock manager. Instead of treating the two components separately, it might be useful to combine the resilient object and lock manager into a single object which handles both concurrency control and resiliency. By removing the need to reason about the interface between the resilient object and the lock manager, this strategy might permit simpler proofs. This remains to be seen, however.

We are currently using the same framework to study many other algorithms. For example, in [7], we consider a variant of the weak concurrent system of this paper, in which the only operations on objects are reads and writes, and in which read and write locks are maintained. The resulting algorithm is essentially that of Moss [22]. We give a description and complete correctness proof. We are currently generalizing the work of the present paper and that of [7], to a "generalized locking algorithm" for nested transactions; this algorithm will take advantage of special properties of the objects to permit flexible patterns of lock sharing. We are also considering timestamp and multiversion algorithms [1], and algorithms which use replicated data objects [8].

We are particularly interested in studying algorithms which give rise to live orphans, i.e., live transactions whose ancestors have aborted [9, 12, 17, 26]. Our serial correctness condition provides a formal definition of *orphan correctness*—that all transactions (including orphans) "see consistent data" [9]. In [11], we describe and prove correctness of several of the recently-developed algorithms for orphan management. This work is simplified by building on the foundation provided by the present paper.

Another direction of interest is the explicit representation of distribution within the model. It is fairly natural to model each transaction and object as located at different sites, each with a local automaton representing the resident portion of the (distributed) scheduler. These automata would communicate with each other in order to implement the (centralized) schedular studied here. The natural next step

would be to model failure resilience, as various components lose information or fail altogether.

The reader might have noted that our correctness conditions do not guarantee anything about the system making progress, but only about "safety" properties. Further work is needed to incorporate guaranatees of progress. This work is likely to be difficult, however. Only recently, in [20], have we achieved what we consider to be a satisfactory understanding of the eventuality and fairness issues for the basic I/O automaton model, so that we can even formulate the conditions we want to satisfy. But even with the ability to state such conditions, the algorithmic issues still seem difficult.

10. Acknowledgment

We thank Bill Weihl for many, many comments and questions, and much encouragement, during the course of this project. We also thank all the other members of the ARGUS design and implementation group at MIT, for providing a concrete model for us to try to abstract and generalize. Also, we thank Yehuda Afek, Alan Fekete, Ken Goldman and Sharon Perl for their comments on earlier drafts of the paper.

References

[1] J. Aspnes, A. Fekete, N. Lynch, M. Merritt and W. Weihl, A theory of timestamp-based concurrency control for nested transactions, in: *Proc. 14th Internat. Conf. on Very Large Data Bases*, Los Angeles, CA (1988).

[2] J.E. Allchin and M.S. McKendry, Synchronization and recovery of actions, in: *Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Montreal, Quebec, Canada (1982) 31-44.

[3] C. Beeri, P.A. Bernstein and N. Goodman, A model for concurrency in nested transaction systems, Tech. Rept. TR CS-86-1, Department of Computer Science, The Hebrew University, Jerusalem, Israel.

[4] C. Beeri, P.A. Bernstein, N. Goodman, M.Y. Lai and D.E. Shasa, A concurrency control theory for nested transactions, in: *Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Montreal, Quebec, Canada (1982) 45-62.

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

[6] K.P. Eswaren, J.N. Gray, R.A. Lorie and I.L. Traiger, The notions of consistency and predicate locks in a data base systems, *Comm. ACM* 19(11) (1976) 624-633.

[7] A. Fekete, N. Lynch, M. Merritt and W. Weihl, Nested transactions and read-write locking, in: *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, San Diego, CA (1987) 97-111.

[8] K. Goldman and N. Lynch, Quorum consensus in nested transaction systems, in: *Proc. 6th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Vancouver, B.C., Canada (1987) 27-41.

[9] J.A. Goree Jr., Internal consistency of a distributed transaction system with orphan detection, M.Sc. Thesis, Technical Rept. MIT/LCS/TR-286, MIT Laboratory for Computer Science, Cambridge, MA, 1983.

[10] J. Gray, Notes on database operating systems, in: R. Bayer, R. Graham and G. Seegmuller, eds., *Operating Systems: an Advanced Course*, Lecture Notes in Computer Science **60** (Springer, Berlin, 1978) 393-481.

[11] M. Herlihy, N. Lynch, M. Merritt and W. Weihl, On the correctness of orphan elimination algorithms, in: *Proc. 17th Internat. Symp. on Fault-Tolerant Computing*, Pittsburgh, PA (1987); also: *J. ACM*, to appear.

[12] M. Herlihy and M. McKendry, Time-driven orphan elimination, in: *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, Los Angeles, CA (1986) 42-48.

[13] C.A.R. Hoare, *Communicating Sequential Processs* (Prentice-Hall, Englewood Cliffs, NJ, 1985).

[14] Z. Kedem and A. Silberschatz, A characterization of database graphs admitting a simple locking protocol, *Acta Inform.* **16** (1981) 1-13.

[15] B.W. Lampson and H.E. Sturgis, Crash recovery in a distributed data storage system, Tech. Rept., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, CA, 1979.

[16] B. Liskov, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl, *Argus Reference Manual*, Programming Methodology Group Memo 54, 1987.

[17] B. Liskov and R. Ladin, Highly-available distributed services and fault-tolerant distributed garbage collection, in: *Proc. 5th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Calgary, Alberta, Canada (1986) 29-39.

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

[19] N.A. Lynch, Concurrency control for resilient nested transactions, *Adv. Comput. Res.* **3** (1986) 335-373.

[20] N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: *Proc. 6th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Vancouver, B.C., Canada (1987) 137-151.

[21] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980).

[22] J.E.B. Moss, Nested transactions: an approach to reliable distributed computing Ph.D. Thesis, Technical Rept. MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA, 1981; also, published by MIT Press, 1985.

[23] D.P. Reed, Naming and synchronization in a decentralized computer system, Ph.D. Thesis, Technical Rept. MIT/LCS/TR-205, MIT Laboratory for Computer Science , Cambridge, MA, 1978.

[24] D.P. Reed, Implementing atomic actions on decentralized data, *ACM Trans. Comput. Systems* **1**(1) (1983) 3-23.

[25] D.J. Rosenkrantz, P.M. Lewis and R.E. Stearns, System level concurrency control for distributed database systems, *ACM Trans. Database Systems* **3**(2) (1978) 178-198.

[26] E.F. Walker, Orphan detection in the Argus system, M.Sc. Thesis, Technical Rept. MIT/LCS/TR-326, MIT Laboratory for Computer Science, Cambridge, MA, 1984.

[27] W.E. Weihl, Specification and implementation of atomic data types, Ph.D. Thesis, Technical Rept. MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA, 1984.

[28] W.E. Weihl, Personal communication, 1986.