A Theory of Atomic Transactions

Nancy Lynch, M.I.T.
Michael Merritt, AT&T Bell Labs
William Weihl, M.I.T.
Alan Fekete, M.I.T.

Abstract:

This paper describes some results of a recent project to develop a theory for reasoning about atomic transactions. This theory allows careful statement of the correctness conditions to be satisfied by transaction-processing algorithms, as well as clear and concise description of such algorithms. It also serves as a framework for rigorous correctness proofs.

1 Introduction

The notion of "atomic transaction", originally introduced for databases, is now used in programming systems for general (data-oriented) distributed computing such as Argus [Liskov] and Camelot [SS]. Roughly speaking, a transaction is a sequence of accesses to data objects; it should execute "as if" it ran with no interruption by other transactions. Moreover, a transaction can complete either successfully or unsuccessfully, by "committing" or "aborting". If it commits, any alterations it makes to the database should be lasting; if it aborts, it should be "as if" it never altered the database at all. The execution of a set of transactions should be "serializable", that is, equivalent to an execution in which no transactions run concurrently and in which all accesses of committed transactions, but no accesses of aborted transactions, are performed. Another condition often considered is "external consistency", which asserts that the order of transactions in the equivalent serial execution should be compatible with the order in which transaction invocations and responses occur.

In order for transactions to be useful for general distributed programming, the notion needs to be extended to include nesting. Thus, in addition to accesses, a transaction can also contain subtransactions. The transaction nesting structure can be described by a forest, with the top-level transactions at the roots and the accesses to data at the leaves. The semantics of nested transactions generalize those of ordinary transactions as follows. Each set of sibling transactions or subtransactions is supposed to execute serializably. As for top-level transactions, subtransactions can commit or abort. Each set of sibling transactions runs as if all the transactions that committed ran in serial order, and all the transactions that aborted did not run at all. An external consistency property is also required for each set of siblings.

Nested transactions provide a very flexible programming mechanism. They allow the programmer to describe more concurrency than would be allowed by single-level transactions, by having transactions request the creation of concurrent subtransactions. They also allow localized handling of transaction failures. When a subtransaction commits or aborts, the commit or abort is reported to its parent transaction. The parent can then decide on its next action based on the reported results. For example, if a subtransaction aborts, its parent can use the reported abort to trigger another subtransaction, one that implements some alternative action. A good mechanism for handling failures is especially important in distributed systems, where failures are common because of the unreliability of communication.

---

[1]The work of the first author (and through her, the work of the fourth author) was supported in part be the office of Naval Research under Contract N00014-85-K-0168, by the National Science Foundation under Grant CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The work of the third author was supported in part by the National Science Foundation under Grant CCR-8716884, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

The idea of nested transactions seems to have originated in the "spheres of control" work of [Davies]. Reed [Reed] developed the current notion of nesting and designed a timestamp-based implementation. Moss [Moss] later designed a locking implementation that serves as the basis of the implementation of the Argus programming language.

There are two reasons why a formal model is needed for reasoning about atomic transactions. First, the implementors of languages that contain transactions need a model in order to reason about the correctness of their implementations. Some of the algorithms that have been proposed for implementing transactions are complicated, and informal arguments about their correctness are unsatisfying. In fact, it is not even obvious how to state the precise correctness conditions to be satisfied by the implementations; a model is needed for describing the semantics of transactions carefully and formally. Second, if programming languages containing transactions become popular, users of these languages will need a model to help them reason about the behavior of their programs.

There has been considerable prior work on a theory for atomic transactions, described, for example, in [BHG]. This "classical" theory is primarily applicable to single-level transactions, rather than nested transactions. It treats both concurrency control and recovery algorithms, although the treatments of the two kinds of algorithms are not completely integrated. The theory assumes a system organization in which accesses are passed from the transactions to a "scheduler", which determines the order in which they are to be performed by the database. The database handles recovery from transaction abort and media failure, so that each access to one data object is performed in the state resulting from all previous non-aborted accesses to that object. The notion of "serializability" in this theory corresponds to "looking like a serial execution, from the point of view of the database". Proofs for some algorithms are presented, primarily based on one main combinatorial theorem, the "Serializability Theorem". This important theorem states that serializability is equivalent to the absence of cycles in a certain graph representing dependencies among transactions.

There are some limitations of this prior work. First, the notion of correctness is quite restrictive, stated as it is in terms of the object boundary in a particular system organization. The object interface that is described is suitable for single-version locking and timestamp algorithms (in the absence of transaction aborts), but it is much less appropriate for other kinds of algorithms. Multi-version algorithms and replicated data algorithms, for example, maintain object information in a form that is very different from the (single-copy latest-value) form used for the simple algorithms, and the appropriate object interface is also very different. The correctness conditions presented for the simple algorithms in [BHG] thus do not apply without change to these other kinds of algorithms. It seems more appropriate, and useful in not unduly restricting possible implementations, to state correctness conditions at the user interface to the system, rather than the object boundary.

Second, the transactions are not modelled explicitly in the earlier work, but rather implicitly, in terms of axioms about their executions. It is sometimes interesting to reason about the control within a transaction, e.g., to describe how the same transaction would behave when it is placed in different systems. Such reasoning is facilitated by an explicit model which clarifies which actions occur under the transaction's control, and which are due to activity of the environment. Furthermore, it will turn out that the "user interface" mentioned above can be modelled by the boundary between the transactions and the rest of the system; in order to state correctness conditions at this boundary, it is useful to have an explicit model for the transactions.

Third, the prior model does not seem to extend well to treat nested transactions. This seems to be primarily because not everything that needs to be described is modelled explicitly. For example, a subtransaction may have been created only because an earlier attempt aborted, so we must model the abort explicitly to capture this dependence.

Our model remedies the deficiencies described above for the earlier model. This improvement does not come for free: our model contains more detail than the earlier model, and may therefore seem more complicated. It seems to us, however, that this extra detail is necessary. In fact, we believe that the extra detail is useful for understanding not just nested transactions, but also ordinary single-level transactions.

We have already used our theory to present and prove correctness of many transaction-processing algorithms, including locking and timestamp-based algorithms for concurrency control, algorithms for managing replicated data and algorithms for managing "orphan" transactions. This work has been presented in research papers [LM, FLMW1, FLMW2, HLMW, GL, AFLMW, Perl], and we are currently writing a book [LMWF] to unify all the work. There is still much that remains to be done, in particular in modelling the very interesting and complex algorithms that have been developed to implement transactions in the presence of crashes that destroy volatile memory.

In this paper, we present some of the basic results of our theory and attempt to compare them to the corresponding results of the classical theory. In particular, we describe the correctness conditions that we use for transaction systems - notions similar to "serializability" but stated in terms of the transaction boundary. We then present our "Serializability Theorem", a general theorem containing a sufficient condition for proving serializability. Although this theorem is more complicated to state than the classical Serializability Theorem, it is similar in spirit: it shows that the existence of a single ordering of transactions that is consistent with the processing of accesses at each object is sufficient to prove serializability. We use our Serializability Theorem elsewhere to prove correctness for locking [FLMW2] and timestamp algorithms [AFLMW], but in this paper, we only present the theorem itself in detail and mention some of its consequences.

The rest of the paper is organized as follows. Section 2 contains an outline of the I/O automaton model, the basic model for concurrent systems that is used for presenting all of our transaction work. Section 3 contains a description of "serial systems", extremely constrained transaction-processing systems that are defined solely for the purpose of stating correctness conditions for more liberal systems. Section 4 contains a description of "simple systems", very unconstrained transaction-processing systems that represent the common features of most transaction-processing systems. Section 5 contains our Serializability Theorem, stated in terms of simple systems. Section 6 contains a discussion of some applications of the Serializability Theorem and Section 7 contains some final remarks.

2 The I/O Automaton Model

In order to reason carefully about complex concurrent systems such as those that implement atomic transactions, it is important to have a simple and clearly-defined formal model for concurrent computation. The model we use for our work is the recently-developed *input/output automaton* model [LT]. Since its introduction, the model has been used for describing and reasoning about several different types of concurrent systems, including network resource allocation algorithms, communication algorithms, concurrent database systems, shared atomic objects, and dataflow architectures. This section contains an introduction to a simple special case of the model that is sufficient for use in this paper.[2]

2.1 Overview of the Model

I/O automata provide an appropriate model for discrete event systems consisting of concurrently-operating components.[3] The components of a discrete event system can be regarded as discrete event systems themselves. Such a system may be "reactive" in the sense that it interacts with its environment in an ongoing manner (rather than, say, simply accepting an input, computing a function of that input and halting).

Each system component is modelled as an "I/O automaton", which is a mathematical object somewhat like a traditional finite-state automaton. However, an I/O automaton need not be finite-state, but can

---

[2]In this paper, we only consider properties of finite executions, and do not consider "liveness" or "fairness" properties.

[3]By a "discrete event system" we mean an entity that undergoes sudden changes that may be named and observed, and through which the system interacts with its environment.

have an infinite state set. The actions of an I/O automaton are classified as either "input", "output" or "internal". This classification is a reflection of a distinction in the system being modelled, between events (such as the receipt of a message) that are caused by the environment, events (such as sending a message) that the component can perform when it chooses and that affect the environment, and events (such as changing the value of a local variable) that a component can perform when it chooses, but that are undetectable by the environment except through their effects on later events. In the model, an automaton generates output and internal actions autonomously, and transmits output actions instantaneously to its environment. In contrast, the automaton's input is generated by the environment and transmitted instantaneously to the automaton. Our distinction between input and other actions is based on who determines when the action is performed: an automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

The fact that our automata are unable to block inputs distinguishes our model from others, such as Hoare's Communication Sequential Processes ("CSP") [Hoare], or Milner's Calculus of Communicating Systems ("CCS") [Milner]. In these models, communication between two components only occurs when both components are willing to communicate. Thus, for example, a sender of a message is blocked until the corresponding receiver is ready to receive the message. In CSP-like models, input blocking is used for two purposes: as a way of eliminating undesirable inputs, and as a way of blocking the activity of the environment. Our model does not have any way of blocking the environment, but does have other ways of coping with unwanted inputs. For example, suppose that we wish to constrain the behavior of an automaton only in case the environment observes certain restrictions on the production of inputs. Instead of requiring the automaton to block the bad inputs, we permit these inputs to occur; however, we may permit the automaton to exhibit arbitrary behavior in case they do. Alternatively, we may require the automaton to detect bad inputs and respond to them with error messages. Thus, we have simple ways of describing input restrictions, without including input-blocking in the model.

I/O automata may be nondeterministic, and indeed the nondeterminism is an important part of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general, since many results about nondeterministic algorithms apply *a fortiori* to all algorithms obtained by restricting the nondeterministic choices. Moreover, the use of nondeterminism helps to avoid cluttering algorithm descriptions and proofs with inessential details. Finally, the uncertainties introduced by asynchrony make nondeterminism an intrinsic property of real concurrent systems, and so an important property to capture in a formal model of such systems.

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

When a system is modelled by an I/O automaton, each possible run of the system is modelled by an "execution", an alternating sequence of states and actions. The possible activity of the system is captured by the set of all possible executions that can be generated by the automaton. However, not all the information contained in an execution is important to a user of the system, or to an environment in which the system is placed. We believe that what is important about the activity of a system is the externally visible events, and not the states or internal events. Thus, we focus on the automaton's "behaviors" — the subsequences of its executions consisting of external (i.e., input and output) actions. We regard a system as suitable for a purpose if any possible sequence of externally-visible events has appropriate characteristics. Thus, in the model, we formulate correctness conditions for an I/O

automaton in terms of properties of the automaton's behaviors.[4]

One convenient way to specify properties of an I/O automaton's behaviors is in terms of another I/O automaton. That is, we can define a particular "specification automaton" B and say that any automaton A is "correct" if it "implements" B, in the sense that each finite behavior of A is also a finite behavior of B. Often, B will be a simple system that is impractical as a real solution because it is too inefficient or uses global information, while A will be a more efficient or distributed algorithm.

The model permits description of the same system at different levels of abstraction. Abstraction mappings can be defined, which describe the relationship between automata that include implementation detail to more abstract automata that suppress some of the detail. Such mappings can be used as aids in correctness proofs for algorithms: if automaton B is an image of automaton A under an appropriate abstraction mapping, then it can be shown that A implements B.

The model allows very careful and readable descriptions of particular concurrent algorithms. We have developed a simple language for describing automata, based on "precondition" and "effect" specifications for actions. This notation has proved sufficient for describing all algorithms we have attempted so far. However, the model does not constrain the user to describe all automata in this manner; for example, the model is general enough to serve also as a formal basis for languages that include more elaborate constructs for sequential flow of control.

The model also allows clear and precise statement of the correctness conditions that an automaton must satisfy in order that the system modelled by the automaton be said to solve a problem; such conditions can be stated independently of any particular proposed solution. As described above, such properties are often conveniently formulated in terms of implemention of a given automaton, but any other method of specifying properties of external behaviors could be used instead. Finally, once both an algorithm and the correctness condition it is supposed to satisfy have been described in the model, it is then possible to use the model as a basis for a rigorous proof that the algorithm satisfies the given conditions.

2.2 Action Signatures

The formal subject matter of this paper is concerned with finite and infinite sequences describing the executions of automata. Usually, we will be discussing sequences of elements from a universal set of *actions*. Since the same action may occur several times in a sequence, it is convenient to distinguish the different occurrences. Thus, we refer to a particular occurrence of an action in a sequence as an *event*.

The actions of each automaton are classified as either "input", "output", or "internal". In the system being modelled, the distinctions are that input actions are not under the system's control, output actions are under the system's control and are externally observable, and internal actions are under the system's control but are not externally observable. In order to describe this classification formally, each automaton comes equipped with an "action signature".

An *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write *in(S)*, *out(S)* and *int(S)* for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions* and *internal actions* of S, respectively. We let *ext(S)* = in(S) $\cup$ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local(S)* = int(S) $\cup$ out(S), and refer to the actions in local(S) as the *locally-controlled actions* of S. Finally, we let acts(S) = in(S) $\cup$ out(S) $\cup$ int(S), and refer to the actions in acts(S) as the *actions* of S. An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature *extsig(S)* = (in(S),out(S),$\emptyset$), i.e., the action signature that is obtained from S by removing the internal actions.

---

[4]This viewpoint differs from that taken in much of the algorithm specification work in the research literature, in which properties of the states are taken to be of primary concern.

2.3 Input/Output Automata

An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of four components:

- an action signature *sig(A)*,
- a set *states(A)* of *states*,
- a nonempty set *start(A)* $\subseteq$ states(A) of *start states*, and
- a transition relation *steps(A)* $\subseteq$ states(A) $\times$ acts(sig(A)) $\times$ states(A), with the property that for every state s' and input action π there is a transition (s',π,s) in steps(A).[5]

Note that the set of states need not be finite. We refer to an element (s',π,s) of steps(A) as a *step* of A. The step (s',π,s) is called an *input step* of A if π is an input action, and *output steps*, *internal steps*, *external steps* and *locally-controlled steps* are defined analogously. If (s',π,s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. The input-enabling property means that an automaton is not able to block input actions.

If A is an automaton, we sometimes write *acts(A)* as shorthand for acts(sig(A)), and likewise for in(A), out(A), etc. An automaton A is said to be *closed* if all its actions are locally-controlled, i.e., if in(A) = $\emptyset$.

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

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched(α)*. We say that β is a *schedule* of A if β is the schedule of an execution of A. We denote the set of schedules of A by *scheds(A)* and the set of finite schedules of A by *finscheds(A)*. We say that a finite schedule β of A *can leave* A *in* state s if there is some finite execution α of A with final state s and with sched(α) = β. The *behavior* of a sequence β of actions in acts(A), denoted by *beh(β)*, is the subsequence of β consisting of actions in ext(A). The *behavior* of an execution fragment α of A, denoted by *beh(α)*, is defined to be beh(sched(α)). We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by *behs(A)* and the set of finite behaviors of A by *finbehs(A)*.

An *extended step* of an automaton A is a triple of the form (s',β,s), where s' and s are in states(A), β is a finite sequence of actions in acts(A), and there is an execution fragment of A having s' as its first state, s as its last state and β as its schedule. (This execution fragment might consist of only a single state, in the case that β is the empty sequence.)

If β is any sequence of actions and Φ is a set of actions, we write $\beta|\Phi$ for the subsequence of β consisting of actions in Φ. If A is an automaton, we write $\beta|A$ for $\beta|$acts(A). We call this the *projection* of β on A. It can be thought of as the portion of β observable by A.

2.4 Composition

I/O automata may be combined by means of a composition operator, as defined in this section. As a preliminary step, we first define composition of action signatures. Let I be an index set that is at most countable. A collection $\{S_i\}_{i\in I}$ of action signatures is said to be *strongly compatible*[6] if for all i, j $\in$ I,

---

[5] I/O automata, as defined in [LT], also include a fifth component, an equivalence relation *part(A)* on local(sig(A)). This component is used for describing fair executions, and is not needed for the results described in this paper.

[6] A weaker notion called "compatibility" is defined in [LT], consisting of the first two of the three given properties only. In this paper, only the stronger notion will be required.

we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is in $acts(S_i)$ for infinitely many i.

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

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

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$,
- $out(S) = \cup_{i \in I} out(S_i)$, and
- $int(S) = \cup_{i \in I} int(S_i)$.

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

Now we define composition of automata. A collection $\{A_i\}_{i \in I}$ of automata is said to be *strongly compatible* if their action signatures are strongly compatible. The *composition* $A = \Pi_{i \in I} A_i$ of a strongly compatible collection of automata $\{A_i\}_{i \in I}$ has the following components:[7]

- $sig(A) = \Pi_{i \in I}\, sig(A_i)$,
- $states(A) = \Pi_{i \in I}\, states(A_i)$,
- $start(A) = \Pi_{i \in I}\, start(A_i)$, and
- steps(A) is the set of triples (s', π, s) such that for all $i \in I$, (a) if $\pi \in acts(A_i)$ then $(s'[i], \pi, s[i]) \in steps(A_i)$, and (b) if $\pi \notin acts(A_i)$ then $s'[i] = s[i]$.[8]

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

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

We close this subsection with some basic results relating executions, schedules and behaviors of a system of automata to those of the automata being composed. The first result says that the projections of executions of a system onto the components are executions of the components, and similarly for schedules, etc.

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

---

[7]Note that the second and third components listed are just ordinary Cartesian products, while the first component uses a previous definition.

[8]We use the notation s[i] to denote the i^{th} component of the state vector s.

Certain converses of the preceding proposition are also true. In particular, we can prove that schedules of component automata can be "patched together" to form a schedule of the composition, and similarly for behaviors. In order to prove these results, we first state two preliminary lemmas, one involving schedules and one involving behaviors, that say that executions of component automata can be patched together to form an execution of the composition.

Lemma 2: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$. Let α_i be an execution of A_i, for all $i \in I$. Suppose β is a sequence of actions in acts(A) such that $\beta | A_i = \text{sched}(\alpha_i)$ for every i. Then there is an execution α of A such that $\beta = \text{sched}(\alpha)$ and $\alpha_i = \alpha | A_i$ for all i.

Lemma 3: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$. Let α_i be an execution of A_i, for all $i \in I$. Suppose β is a sequence of actions in ext(A) such that $\beta | A_i = \text{beh}(\alpha_i)$ for every i. Then there is an execution α of A such that $\beta = \text{beh}(\alpha)$ and $\alpha_i = \alpha | A_i$ for all i.

Now the results about patching together schedules and patching together behaviors follow easily.

Proposition 4: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$.

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

Proof: By Lemmas 2 and 3.

Proposition 4 provides a method for showing that certain sequences are behaviors of a composition A: first show that its projections are behaviors of the components of A and then appeal to Proposition 4.

2.5 Correspondences Between Automata

In this subsection, we define the notion of "implementation" which is useful in stating correctness conditions to be satisfied by automata. Let A and B be automata with the same external action signature, i.e., with extsig(A) = extsig(B). Then A is said to *implement* B if $\text{finbehs}(A) \subseteq \text{finbehs}(B)$. One reason for the usefulness of the notion of implementation as a correctness condition is the following fact: if A implements B, then replacing B by A in any system yields a new system in which all finite behaviors are behaviors of the original system. In fact, as the following proposition shows, we can take any collection of components of a system and replace each by an implementation, and the resulting system will implement the original one.

Proposition 5: Suppose that $\{A_i\}_{i \in I}$ is a strongly compatible collection of automata, and let $A = \Pi_{i \in I} A_i$. Also suppose that $\{B_i\}_{i \in I}$ is a strongly compatible collection of automata, and let $B = \Pi_{i \in I} B_i$. If for each index i in I, A_i implements B_i, then A implements B.

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

1. For every start state s_0 of A, there is a start state t_0 of B such that $t_0 \in f(s_0)$.

2. Let s' be a reachable state of A, t' $\in$ f(s') a reachable state of B, and (s',π,s) a step of A. Then there is an extended step, (t',γ,t), of B (possibly having an empty schedule) such that the following conditions are satisfied:

   a. $\gamma|ext(B) = \pi|ext(A)$, and

   b. $t \in f(s)$.

Proposition 6: Suppose that A and B are automata with the same external action signature and there is a possibilities mapping, f, from A to B. Then A implements B.

2.6 Preserving Properties

Although an automaton in our model is unable to block input actions, it is often convenient to restrict attention to those behaviors in which the environment provides inputs in a "sensible" way, that is, where the interaction between the automaton and its environment obeys certain "well-formedness" restrictions. A useful way of discussing such restrictions is in terms of the notion that an automaton "preserves" a property of behaviors: as long as the environment does not violate the property, neither does the automaton. Such a notion is primarily interesting for properties that are "prefix-closed" and "limit-closed": formally, a set of sequences P is *prefix-closed* provided that whenever $\beta \in P$ and γ is a prefix of β, it is also the case that $\gamma \in P$. A set of sequences P is *limit-closed* provided that any sequence all of whose finite prefixes are in P is also in P.

Let Φ be a set of actions and P be a nonempty, prefix-closed, limit-closed set of sequences of actions in Φ (i.e., a nonempty, prefix-closed, limit-closed "property" of such sequences). Let A be an automaton with $\Phi \subseteq ext(A)$. We say that A *preserves* P if $\beta\pi \in finbehs(A)$, $\pi \in out(A)$ and $\beta|\Phi \in P$ together imply that $\beta\pi|\Phi \in P$. Thus, if an automaton preserves a property P, the automaton is not the first to violate P: as long as the environment only provides inputs such that the cumulative behavior satisfies P, the automaton will only perform outputs such that the cumulative behavior satisfies P. Note that the fact that an automaton A preserves a property P does not imply that all of A's behaviors, when restricted to Φ, satisfy P; it is possible for a behavior of A to fail to satisfy P, if an input causes a violation of P. However, the following proposition gives a way to deduce that all of a system's behaviors satisfy P. The lemma says that, under certain conditions, if all components of a system preserve P, then all the behaviors of the composition satisfy P.

Proposition 7: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and suppose that A, the composition, is a closed system. Let $\Phi \subseteq ext(A)$, and let P be a nonempty, prefix-closed, limit-closed set of sequences of actions in Φ. Suppose that for each $i \in I$, one of the following is true.

1. $\Phi \subseteq ext(A_i)$ and A_i preserves P, or
2. $\Phi \cap ext(A_i) = \emptyset$.

If $\beta \in behs(A)$, then $\beta|\Phi \in P$.

3 Serial Systems and Correctness

In this section, we develop the formal machinery needed to define correctness for transaction-processing systems. Correctness is expressed in terms of a particular kind of system called a "serial system". We define serial systems here, using I/O automata.

3.1 Overview

Transaction-processing systems consist of user-provided transaction code, plus transaction-processing algorithms designed to coordinate the activities of different transactions. The transactions are written by application programmers in a suitable programming language. In some transaction-processing systems such as the Argus system, transactions have a nested structure, so that transactions can invoke subtransactions and receive responses from the subtransactions describing the results of their processing. In addition to invoking subtransactions, transactions can also invoke operations on data objects.

In a transaction-processing system, the transaction-processing algorithms interact with the transactions, making decisions about when to schedule the creation of subtransactions and the performance of operations on objects. In order to carry out such scheduling, the transaction-processing algorithms may manipulate locks on objects, multiple copies of objects, and other convenient data structures. One popular organization divides the transaction processing into a "scheduler algorithm" and a "database" of objects. In this organization, the scheduler has the power to decide when operations are to be performed on the objects in the database, but not to perform more complex manipulations on objects (such as maintaining multiple copies). Although this organization is popular, it does not encompass all the useful system designs.

In our work, each component of a transaction-processing system is modelled as an I/O automaton. In particular, each transaction is an automaton, and all the transaction-processing algorithms together comprise another automaton.

It is not obvious at first how one ought to model the nested structure of transactions within the I/O automaton model. One might consider defining special kinds of automata that have a nested structure, for example. However, it appears that the cleanest way to model this structure is to describe each subtransaction in the transaction nesting structure as a separate automaton. If a parent transaction T wishes to invoke a child transaction T', T issues an output action that "requests that T' be created". The transaction-processing algorithms receive this request, and at some later time might decide to issue an action that is an input to the child T' and corresponds to the "creation" of T'. Thus, the different transactions in the nesting structure comprise a forest of automata, communicating with each other indirectly through the transaction-processing automaton. The highest-level user-defined transactions, i.e., those that are not subtransactions of any other user-defined transactions, are the roots in this forest.

It is actually more convenient to model the transaction nesting structure as a tree than a forest. Thus, we add an extra "root" automaton as a sort of "dummy transaction", located at the top of the transaction nesting structure. The highest-level user-defined transactions are considered to be children of this new root. The root can be thought of as modelling the outside world, from which invocations of top-level transactions originate and to which reports about the results of such transactions are sent. We often find that the formal reasoning we want to do about this dummy root transaction is very similar to our reasoning about ordinary transactions; thus, regarding the root as a transaction leads to economy in our formal arguments.

The primary goal of this section is to define correctness conditions to be satisfied by transaction-processing systems. As we discussed in the introduction, it seems most natural and general to define correctness conditions in terms of the actions occurring at the boundary between the transactions (including the root transaction) and the transaction-processing automaton. For it is immaterial how the transaction-processing algorithms work, as long as the outside world and the transactions see "correct" behavior. We define correct behavior for a transaction-processing system in terms of the behavior of a particular and very constrained "serial" transaction-processing system, which processes all transactions serially.

Serial systems consist of transaction automata and "serial object automata" composed with a "serial scheduler automaton". Transaction automata have already been discussed above. Serial object automata serve as specifications for permissible object behavior. They describe the responses the objects should make to arbitrary sequences of operation invocations, assuming that later invocations wait for responses to previous invocations. Serial objects are very much like the ordinary abstract data objects that are used in sequential programming languages.

The serial scheduler handles the communication among the transactions and serial objects, and thereby controls the order in which the transactions take steps. It ensures that no two sibling transactions are active concurrently — that is, it runs each set of sibling transactions serially. The serial scheduler is also responsible for deciding if a transaction commits or aborts. The serial scheduler can permit a transaction to abort only if its parent has requested its creation, but it has not actually been created. Thus, in a serial system, all sets of sibling transactions are run serially, and in such a way that

no aborted transaction ever performs any steps.

A serial system would not be an interesting transaction-processing system to implement. It allows no concurrency among sibling transactions, and has only a very limited ability to cope with transaction failures. However, we are not proposing serial systems as interesting implementations; rather, we use them exclusively as specifications for correct behavior of other, more interesting systems. In our work, we describe many systems that do allow concurrency and recovery from transaction failures. (That is, they undo the effects of aborted transactions that have performed significant activity.) We prove that these systems are correct in the sense that certain transactions, and in particular T_0, cannot distinguish them from corresponding serial systems. It appears to the transactions as if all siblings are run serially, and aborted transactions are never created, even though in reality, the systems allow concurrency and recovery from transaction failures.

In the remainder of this section, we develop the necessary machinery for defining serial systems and correctness. First, we define a type structure used to name transactions and objects. Then we describe the general structure of a serial system — the components it includes, the actions the components perform, and the way the components are interconnected. We define several concepts involving the actions of a serial system. We then go on to define the components of a serial system in detail, and state some basic properties of serial systems. Finally, we use serial systems to state correctness conditions for transaction-processing systems.

3.2 System Types

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

A *system type* consists of the following:

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

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

The transaction tree describes the nesting structure for transaction names, with T_0 as the name of the dummy "root transaction". Each child node in this tree represents the name of a subtransaction of the transaction named by its parent. The children of T_0 represent names of the top-level user-defined transactions. The accesses represent names for the lowest-level transactions in the transaction nesting structure; we will use these lowest-level transactions to model operations on data objects. Thus, the only transactions that access data directly are the leaves of the transaction tree. The internal nodes model transactions whose function is to create and manage subtransactions, but not to access data directly.

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

Classical concurrency control theory considers transactions having a simple nesting structure. As modelled in our framework, that nesting structure has three levels; the top level consists of the root T_0, modelling the outside world, the next level consists of all the user-defined transactions, and the lowest level consists of the accesses to data objects.

The set $\mathcal{X}$ is the set of names for the objects used in the system. Each access transaction name denotes an access to some particular object, as designated by the "object" mapping. If $X \in \mathcal{X}$, the set of accesses T for which object(T) = X is called *accesses(X)*.

The set $\mathcal{V}$ of return values is the set of possible values that might be returned by successfully-completed transactions to their parents. If T is an access transaction name, and v is a return value, we say that the pair (T,v) is an *operation* of the given system type. Thus, an operation designates a particular access to an object and a particular value returned by the access.

3.3 General Structure of Serial Systems

A serial system for a given system type is a closed system consisting of a "transaction automaton" A(T) for each non-access transaction name T, a "serial object automaton" S(X) for each object name X, and a single "serial scheduler automaton". Later in this chapter, we will give a precise definition for the serial scheduler automaton, and will give conditions to be satisfied by the transaction and object automata. Here, we just describe the signatures of the various automata, in order to explain how the automata are interconnected.

The following diagram depicts the structure of a serial system.

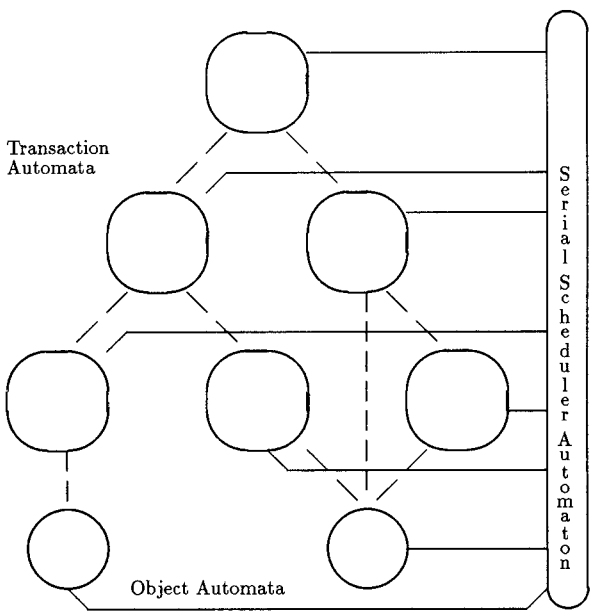

Figure 1: Serial System Structure

The transaction nesting structure is indicated by dotted lines, and the direct connections between automata (via shared actions) are indicated by solid lines. Thus, the transaction automata interact directly with the serial scheduler, but not directly with each other or with the object automata. The object automata also interact directly with the serial scheduler.

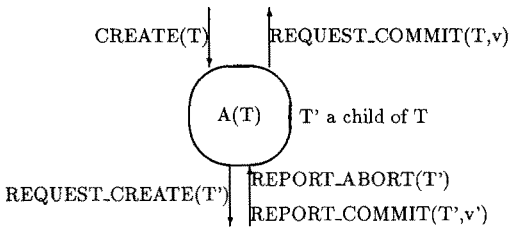

Figure 2: Transaction Automaton

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

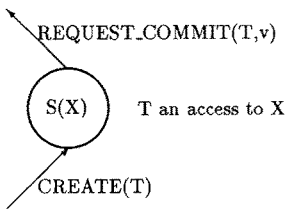

Figure 3: Object Automaton

Figure 3 shows the object interface. Object X has input CREATE(T) actions for each T in accesses(X). These actions should be thought of as invocations of operations on object X. Object X also has output actions of the form REQUEST_COMMIT(T,v), representing responses to the invocations. The value v in a REQUEST_COMMIT(T,v) action is a return value returned by the object as part of its response. We have chosen to use the "create" and "request_commit" notation for the object actions, rather than the more familiar "invoke" and "respond" terminology, in the interests of uniformity: there are many places in our formal arguments where access transactions can be treated uniformly with non-access transactions, and so it is useful to have a common notation for them.

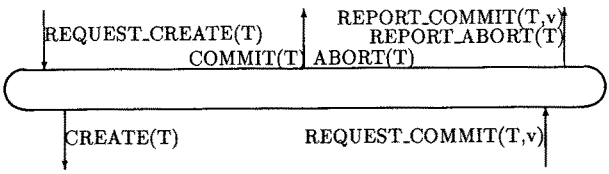

Figure 4: Serial Scheduler Automaton

Figure 4 shows the serial scheduler interface. The serial scheduler receives the previously-mentioned REQUEST_CREATE and REQUEST_COMMIT actions as inputs from the other system components. It produces CREATE actions as outputs, thereby awakening transaction automata or invoking operations on objects. It also produces COMMIT(T) and ABORT(T) actions for arbitrary transactions $T \neq T_0$, representing decisions about whether the designated transaction commits or aborts. For technical convenience, we classify the COMMIT and ABORT actions as output actions of the serial scheduler, even though they are not inputs to any other system component. Finally, the serial scheduler has REPORT_COMMIT and REPORT_ABORT actions as outputs, by which it communicates the fates of transactions to their parents.

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

3.4 Serial Actions

The *serial actions* for a given system type are defined to be the external actions of a serial system of that type. These are just the actions listed in the preceding section: CREATE(T) and REQUEST_COMMIT(T,v), where T is any transaction name and v is a return value, and REQUEST_CREATE(T), COMMIT(T), ABORT(T), REPORT_COMMIT(T,v), and REPORT_ABORT(T) where $T \neq T_0$ is a transaction name and v is a return value.

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

3.4.1 Basic Definitions

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

We define the "transaction" of an action that appears in the interface of a transaction or object automaton — that is, of any non-completion action. Let T be any transaction name. If π is one of the serial actions CREATE(T), REQUEST_COMMIT(T,v), or REQUEST_CREATE(T'), REPORT_COMMIT(T',v') or REPORT_ABORT(T'), where T' is a child of T, then we define *transaction*(π) to be T. If π is a completion action, then transaction(π) is undefined. We will sometimes want to associate a transaction with completion actions as well as other serial actions; since a completion action for T can be thought of as occurring "in between" T and parent(T), we will sometimes want to associate T and sometimes parent(T) with the action. Thus, we extend the "transaction(π)"

definition in two different ways. If π is any serial action, then we define *hightransaction(π)* to be transaction(π) if π is not a completion action, and to be parent(T), if π is a completion action for T. Also, if π is any serial action, we define *lowtransaction(π)* to be transaction(π) if π is not a completion action, and to be T, if π is a completion action for T. In particular, hightransaction(π) = lowtransaction(π) = transaction(π) for all serial actions other than completion actions.

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

We extend the notation in the preceding paragraphs to events as well as actions. For example, if π is an event, then we write *transaction(π)* to denote the transaction of the action of which π is an occurrence.

Recall that an *operation* is a pair (T,v), consisting of a transaction name and a return value. We can associate operations with a sequence of serial actions, as follows. If β is a sequence of serial actions, we say that the operation (T,v) *occurs* in β if there is a REQUEST_COMMIT(T,v) event in β. Conversely, we can associate serial actions with a sequence of operations. For any operation (T,v), let *perform(T,v)* denote the two-action sequence CREATE(T) REQUEST_COMMIT(T,v), the expansion of (T,v) into its two parts. This definition is extended to sequences of operations in the natural way: if ξ is a sequence of operations of the form ξ'(T,v), then perform(ξ) = perform(ξ') perform(T,v). Thus, the "perform" function expands a sequence of operations into a corresponding alternating sequence of CREATE and REQUEST_COMMIT actions.

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

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

Sometimes we will want to use definitions from this subsection for sequences of actions chosen from some other set besides the set of serial actions — usually, a set containing the set of serial actions. We extend the appropriate definitions of this subsection to such sequences by applying them to the subsequences consisting of serial actions. Thus, if β is a sequence of actions chosen from a set Φ of actions, define *serial(β)* to be the subsequence of β consisting of serial actions. Then we say that operation (T,v) *occurs* in β exactly if it occurs in serial(β). A transaction T is said to be *active* in β provided that it is active in serial(β), and similarly for the "live" and "orphan" definitions. Also, $\beta|\mathcal{U}$ is defined to be serial(β)$|\mathcal{U}$, and similarly for projection on an object.

3.4.2 Well-Formedness

In the definition of a serial system in the following subsection, we will place very few constraints on the transaction automata and serial object automata. However, we will want to assume that certain simple properties are guaranteed; for example, a transaction should not take steps until it has been created, and an object should not respond to an operation that has not been invoked. Such requirements are captured by "well-formedness conditions", certain properties of sequences of external actions of the transaction and object components. We define those conditions here.

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

conditions hold.

1. The first event in β, if any, is a CREATE(T) event, and there are no other CREATE events.
2. There is at most one REQUEST_CREATE(T') event in β for each child T' of T.
3. Any report event for a child T' of T is preceded by REQUEST_CREATE(T') in β.
4. There is at most one report event in β for each child T' of T.
5. If a REQUEST_COMMIT event for T occurs in β, then it is preceded by a report event for each child T' of T for which there is a REQUEST_CREATE(T') in β.
6. If a REQUEST_COMMIT event for T occurs in β, then it is the last event in β.

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

Now we define "serial object well-formedness". Let X be any object name. A sequence of serial actions π with object(π) = X is defined to be *serial object well-formed* for X if it is a prefix of a sequence of the form CREATE(T_1) REQUEST_COMMIT(T_1,v_1) CREATE(T_2) REQUEST_COMMIT(T_2,v_2) ..., where $T_i \neq T_j$ when $i \neq j$. The following connection between serial object well-formedness and transaction well-formedness is immediate.

Lemma 8: Let β be a sequence of serial actions π with object(π) = X. If β is serial object well-formed for X and T is an access to X, then $\beta|T$ is transaction well-formed for T.

3.5 Serial Systems

We are now ready to define "serial systems". Serial systems are composed of transaction automata, serial object automata, and a single serial scheduler automaton. There is one transaction automaton A(T) for each non-access transaction name T, and one serial object automaton S(X) for each object name X. We describe the three kinds of components in turn.

3.5.1 Transaction Automata

A *transaction automaton* A(T) for a non-access transaction name T of a given system type is an I/O automaton with the following external action signature.

Input:
  CREATE(T)
  REPORT_COMMIT(T',v), for T' a child of T, and v a return value
  REPORT_ABORT(T'), for T' a child of T
Output:
  REQUEST_CREATE(T'), for T' a child of T
  REQUEST_COMMIT(T,v), for v a return value

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

Transaction automata are intended to be general enough to model the transactions defined in any reasonable programming language. Of course, there is still work required in showing how to define appropriate transaction automata for the transactions in any particular language. This correspondence depends on the special features of each language, and we do not describe techniques for establishing such a correspondence in this paper.

3.5.2 Serial Object Automata

A *serial object automaton* S(X) for an object name X of a given system type is an I/O automaton with the following external action signature.

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

In addition, S(X) may have an arbitrary set of internal actions. We require S(X) to preserve serial object well-formedness for X, as defined in the previous section. As with transaction automata, serial object automata can be chosen arbitrarily as long as they preserve serial object well-formedness.

3.5.3 Serial Scheduler

There is a single serial scheduler automaton for each system type. It runs transactions according to a depth-first traversal of the transaction tree, running sets of sibling transactions serially. 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) action are that transaction T was never created. The scheduler does not permit any two sibling transactions to be live at the same time, and does not abort any transaction while any of its siblings is active. We now give a formal definition of the serial scheduler automaton.

The action signature of the serial scheduler is as follows.

Input:
REQUEST_CREATE(T), $T \neq T_0$
REQUEST_COMMIT(T,v)
Output:
CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T,v), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$

Each state s of the serial scheduler consists of six sets, denoted via record notation: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted and s.reported. The set s.commit_requested is a set of operations. The others are sets of transactions. There is exactly one start state, in which the set create_requested is $\{T_0\}$, and the other sets are empty. We use the notation s.completed to denote s.committed $\cup$ s.aborted. Thus, s.completed is not an actual variable in the state, but rather a "derived variable" whose value is determined as a function of the actual state variables.

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

REQUEST_CREATE(T), $T \neq T_0$
Effect:
s.create_requested = s'.create_requested $\cup$ {T}

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

CREATE(T)
Precondition:
$T \in$ s'.create_requested - s'.created
$T \notin$ s'.aborted

siblings(T) $\cap$ s'.created $\subseteq$ s'.completed
Effect:
s.created = s'.created $\cup$ {T}

COMMIT(T), $T \neq T_0$
Precondition:
(T,v) $\in$ s'.commit_requested for some v
$T \notin$ s'.completed
Effect:
s.committed = s'.committed $\cup$ {T}

ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.create_requested - s'.completed
$T \notin$ s'.created
siblings(T) $\cap$ s'.created $\subseteq$ s'.completed
Effect:
s.aborted = s'.aborted $\cup$ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
$T \in$ s'.committed
(T,v) $\in$ s'.commit_requested
$T \notin$ s'.reported
Effect:
s.reported = s'.reported $\cup$ {T}

REPORT_ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.aborted
$T \notin$ s'.reported
Effect:
s.reported = s'.reported $\cup$ {T}

Thus, the input actions, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded. A CREATE action can occur only if a corresponding REQUEST_CREATE has occurred and the CREATE has not already occurred. Moreover, it cannot occur if the transaction was previously aborted. The third precondition on the CREATE action says that the serial scheduler does not create a transaction until each of its previously created sibling transactions has completed (i.e., committed or aborted). That is, siblings are run sequentially. A COMMIT action can occur only if it has previously been requested and no completion action has yet occurred for the indicated transaction. An ABORT action can occur only if a corresponding REQUEST_CREATE has occurred and no completion action has yet occurred for the indicated transaction. Moreover, it cannot occur if the transaction was previously created. The third precondition on the ABORT action 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 dealt with sequentially with respect to their siblings. The result of a transaction can be reported to its parent at any time after the commit or abort has occurred.

The following lemma describes simple relationships between the state of the serial scheduler and its computational history.

Lemma 9: Let β be a finite schedule of the serial scheduler, and let s be a state such that β can leave the serial scheduler in state s. Then the following conditions are true.

1. $T \in$ s.create_requested exactly if $T = T_0$ or β contains a REQUEST_CREATE(T) event.
2. $T \in$ s.created exactly if β contains a CREATE(T) event.

3. (T,v) $\in$ s.commit_requested exactly if β contains a REQUEST_COMMIT(T,v) event.
4. T $\in$ s.committed exactly if β contains a COMMIT(T) event.
5. T $\in$ s.aborted exactly if β contains an ABORT(T) event.
6. T $\in$ s.reported exactly if β contains a report event for T.
7. s.committed $\cap$ s.aborted = $\emptyset$.
8. s.reported $\subseteq$ s.committed $\cup$ s.aborted.

The following lemma gives simple facts about the actions appearing in an arbitrary schedule of the serial scheduler.

Lemma 10: Let β be a schedule of the serial scheduler. Then all of the following hold:

1. If a CREATE(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
2. At most one CREATE(T) event appears in β for each transaction T.
3. If a COMMIT(T) event appears in β then a REQUEST-COMMIT(T,v) event precedes it in β for some return value v.
4. If an ABORT(T) event appears in β then a REQUEST-CREATE(T) event precedes it in β.
5. If a CREATE(T) or ABORT(T) event appears in β and is preceded by a CREATE(T') event for a sibling T' of T, then it is also preceded by a completion event for T'.
6. At most one completion event appears in β for each transaction.
7. At most one report event appears in β for each transaction.
8. If a REPORT-COMMIT(T,v) event appears in β, then a COMMIT(T) event and a REQUEST_COMMIT(T,v) event precede it in β.
9. If a REPORT-ABORT(T) event appears in β, then an ABORT(T) event precedes it in β.

Proof: By Lemma 9 and the serial scheduler preconditions.

The final lemma of this subsection says that the serial scheduler preserves the well-formedness properties described earlier.

Lemma 11:

1. Let T be any transaction name. Then the serial scheduler preserves transaction well-formedness for T.
2. Let X be any object name. Then the serial scheduler preserves serial object well-formedness for X.

Proof: By the definitions and the characterization given in Lemma 10.

3.5.4 Serial Systems, Executions, Schedules and Behaviors

A *serial system* of a given system type is the composition of a strongly compatible set of automata indexed by the union of the set of non-access transaction names, the set of object names and the singleton set {SS} (for "serial scheduler"). Associated with each non-access transaction name T is a transaction automaton A(T) for T. Associated with each object name X is a serial object automaton S(X) for X. Finally, associated with the name SS is the serial scheduler automaton for the given system type. When the particular serial system is understood from context, we will sometimes use the terms *serial executions*, *serial schedules* and *serial behaviors* for the system's executions, schedules and behaviors, respectively.

A fundamental property of serial behaviors is that they are well-formed for each transaction and object name.

Proposition 12: If β is a serial behavior, then the following conditions hold.

1. For every transaction name T, $\beta|T$ is transaction well-formed for T.
2. For every object name X, $\beta|X$ is serial object well-formed for X.

Proof: For non-access transaction names T, or arbitrary object names X, the result is immediate by Lemma 7, the definitions of transaction and object automata, and Lemma 11. Suppose that T is an access to X. Since $\beta|X$ is serial object well-formed for X, Lemma 8 implies that $\beta|T$ is transaction well-formed for T.

Another fundamental property of serial behaviors is that the live transactions always form a chain of ancestors, as indicated below.

Proposition 13: Let β be a serial behavior.

1. If T is live in β and T' is an ancestor of T, then T' is live in β.
2. If T and T' are transaction names such that both T and T' are live in β, then either T is an ancestor of T' or T' is an ancestor of T.

In the remainder of the paper, we fix an arbitrary system type and serial system, with A(T) as the non-access transaction automaton for each transaction name T, and S(X) as the serial object automaton for each object name X.

3.6 Correctness Conditions

Now that we have defined serial systems, we can use them to state correctness conditions for other transaction-processing systems. It is reasonable to use serial systems in this way because of the particular constraints the serial scheduler imposes on the orders in which transactions and objects can perform steps. We contend that the given constraints correspond precisely to the way nested transaction systems ought to appear to behave; in particular, these constraints yield a natural generalization of the notion of serial execution in classical transaction systems. We arrive at a number of correctness conditions by considering *for which system components* this appearance must be maintained: for the external environment T_0, for all transactions, or for all non-orphan transactions.

To express these correctness conditions we define the notion of "serial correctness" of a sequence of actions for a particular transaction name. We say that a sequence β of actions is *serially correct* for transaction name T provided that there is some serial behavior γ such that $\beta|T = \gamma|T$. (Recall that if T is a non-access, we have $\beta|T = \beta|ext(A(T))$ and $\gamma|T = \gamma|ext(A(T))$. If T is a non-access transaction, serial correctness for T is a condition that guarantees to implementors of T that their code will encounter only situations that can arise in serial executions.

The principal notion of correctness that we will use in this paper is the serial correctness of all finite behaviors for the root transaction name T_0. This says that the "outside world" cannot distinguish between the given system and the serial system.

Many of the algorithms we study satisfy stronger correctness conditions. A fairly strong and possibly interesting correctness condition is the serial correctness of all finite behaviors for all transactions names. Thus, neither the outside world nor any of the individual user transactions can distinguish between the given system and the serial system. Note that the definition of serial correctness for all transactions does not require that all the transactions see behavior that is part of the same execution of the serial system; rather, each could see behavior arising in a different execution.

We will also consider intermediate conditions such as serial correctness for all non-orphan transaction names. This condition implies serial correctness for T_0 because the serial scheduler does not have the action $ABORT(T_0)$ in its signature, so T_0 cannot be an orphan. Most of the popular algorithms for concurrency control and recovery guarantee serial correctness for all non-orphan transaction names. Our Serializability Theorem gives sufficient conditions for showing that a behavior of a transaction-processing system is serially correct for an arbitrary non-orphan transaction name, and can be used to prove this property for many of these algorithms. The usual algorithms do not guarantee serial

correctness for orphans, however; in order to guarantee this as well, the use of a special "orphan management" algorithm is generally required. Such algorithms are described and their correctness proved in [HLMW].

We close this subsection with a proposition that shows that serial correctness with respect to a transaction name T, a notion defined in terms of behaviors of T, implies a relationship between executions of T in the two systems.

Proposition 14: Let $\{B_i\}_{i \in I}$ be a strongly compatible set of automata and let $B = \Pi_{i \in I} B_i$. Suppose that non-access transaction name T is in the index set I and suppose that B_T and A(T) are the same automaton. Let α be a finite execution of B, and suppose that beh(α) is serially correct for T. Then there is a serial execution α' such that $\alpha | B_T = \alpha' | A(T)$.

Proof: Proposition 1 implies that $\alpha | B_T$ is an execution of B_T, and then Lemma 3 can be used to patch together the desired execution.

4 Simple Systems

It is desirable to state our Serializability Theorem in such a way that it can be used for proving correctness of many different kinds of transaction-processing systems, with radically different architectures. We therefore define a "simple system", which embodies the common features of most transaction-processing systems, independent of their concurrency control and recovery algorithms, and even of their division into modules to handle different aspects of transaction-processing. A "simple system" consists of the transaction automata together with a special automaton called the "simple database". Our theorem is stated in terms of simple systems.

Many complicated transaction-processing algorithms can be understood as implementations of the simple system. For example, a system containing separate objects that manage locks and a "controller" that passes information among transactions and objects can be represented in this way, and so our theorem can be used to prove its correctness. The same strategy works for a system containing objects that manage timestamped versions and a controller that issues timestamps to transactions.

4.1 Simple Database

There is a single simple database for each system type. The action signature of the simple database is that of the composition of the serial scheduler with the serial objects:

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

States of the simple database are the same as for the serial scheduler, and the initial states are also the same. The transition relation is as follows.

REQUEST_CREATE(T), $T \neq T_0$
Effect:
  s.create_requested = s'.create_requested $\cup$ {T}

REQUEST_COMMIT(T,v), T a non-access
Effect:
  s.commit_requested = s'.commit_requested $\cup$ {(T,v)}

CREATE(T)
Precondition:
$T \in$ s'.create_requested - s'.created
Effect:
s.created = s'.created $\cup$ {T}

COMMIT(T), $T \neq T_0$
Precondition:
(T,v) $\in$ s'.commit_requested for some v
$T \notin$ s'.completed
Effect:
s.committed = s'.committed $\cup$ {T}

ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.create_requested - s'.completed
Effect:
s.aborted = s'.aborted $\cup$ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
$T \in$ s'.committed
(T,v) $\in$ s'.commit_requested
$T \notin$ s'.reported
Effect:
s.reported = s'.reported $\cup$ {T}

REPORT_ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.aborted
$T \notin$ s'.reported
Effect:
s.reported = s'.reported $\cup$ {T}

REQUEST_COMMIT(T,v), T an access
Precondition:
$T \in$ s'.created
for all v', (T,v') $\notin$ s'.commit_requested
Effect:
s.commit_requested = s'.commit_requested $\cup$ {(T,v)}

The next two lemmas are analogous to those previously given for the serial scheduler.

Lemma 15: Let β be a finite schedule of the simple database, and let s be a state such that β can leave the simple database in state s. Then the following conditions are true.

1. T is in s.create_requested exactly if $T = T_0$ or β contains a REQUEST_CREATE(T) event.
2. T is in s.created exactly if β contains a CREATE(T) event.
3. (T,v) is in s.commit_requested exactly if β contains a REQUEST_COMMIT(T,v) event.
4. T is in s.committed exactly if β contains a COMMIT(T) event.
5. T is in s.aborted exactly if β contains an ABORT(T) event.
6. T is in s.reported exactly if β contains a report event for T.
7. s.committed $\cap$ s.aborted = $\emptyset$.
8. s.reported $\subseteq$ s.committed $\cup$ s.aborted.

Lemma 16: Let β be a schedule of the simple database. Then all of the following hold:

1. If a CREATE(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
2. At most one CREATE(T) event appears in β for each transaction T.
3. If a COMMIT(T) event appears in β, then a REQUEST-COMMIT(T,v) event precedes it in β for some return value v.
4. If an ABORT(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
5. At most one completion event appears in β for each transaction.
6. At most one report event appears in β for each transaction.
7. If a REPORT-COMMIT(T,v) event appears in β, then a COMMIT(T) event and a REQUEST_COMMIT(T,v) event precede it in β.
8. If a REPORT-ABORT(T) event appears in β, then an ABORT(T) event precedes it in β.
9. If T is an access and a REQUEST_COMMIT(T,v) event occurs in β, then a CREATE(T) event precedes it in β.
10. If T is an access, then at most one REQUEST_COMMIT event for T occurs in β.

Proof: By Lemma 15 and the simple database preconditions.

Thus, the simple database embodies those constraints that we would expect any reasonable transaction-processing system to satisfy. The simple database does not allow CREATEs, ABORTs, or COMMITs without an appropriate preceding request, does not allow any transaction to have two creation or completion events, and does not report completion events that never happened. Also, it does not produce responses to accesses that were not invoked, nor does it produce multiple responses to accesses. On the other hand, the simple database allows almost any ordering of transactions, allows concurrent execution of sibling transactions, and allows arbitrary responses to accesses. We do not claim that the simple database produces only serially correct behaviors; rather, we use the simple database to model features common to more sophisticated systems that do ensure correctness.

Lemma 17: Let T be any transaction name. Then the simple database preserves transaction well-formedness for T.

Proof: By the definitions and the characterization given in Lemma 16.

4.2 Simple Systems, Executions, Schedules and Behaviors

A *simple system* is the composition of a compatible set of automata indexed by the union of the set of non-access transaction names and the singleton set {SD} (for "simple database"). Associated with each non-access transaction name T is a transaction automaton A(T) for T, and associated with the name SD is the simple database automaton for the given system type. When the particular simple system is understood from context, we will often use the terms *simple executions*, *simple schedules* and *simple behaviors* for the system's executions, schedules and behaviors, respectively.

Lemma 18: If β is a simple behavior and T is a transaction name, then β|T is transaction well-formed for T.

Proof: By Lemma 17 and the definition of transaction automata.

The Serializability Theorem is formulated in terms of simple behaviors; it provides a sufficient condition for a simple behavior to be serially correct for a particular transaction name T.

5 The Serializability Theorem

In this section, we present our Serializability Theorem, which embodies a fairly general method for proving that a concurrency control algorithm guarantees serial correctness. This theorem expresses the following intuition: a behavior of a system is serially correct provided that there is a way to order the transactions so that when the operations at each object are arranged in the corresponding order, the result is a behavior of the corresponding serial object. The correctness of many different concurrency control algorithms can be proved using this theorem.

This theorem is the closest analog we have for the classical Serializability Theorem of [BHG]. Both that theorem and ours hypothesize that there is some ordering on transactions consistent with the behavior at each object. In both cases, this hypothesis is used to show serial correctness. Our result is somewhat more complicated, however, because it deals with nesting and aborts. In the next two subsections, we give some additional definitions that are needed to accomodate these complications.

5.1 Visibility

One difference between our result and the classical Serializability Theorem is that the conclusion of our result is serial correctness for an arbitrary transaction T, whereas the classical result essentially considers only serial correctness for T_0. Thus, it should not be surprising that the hypothesis of our result does not deal with all the operations at each object, but only with those that are in some sense "visible" to the particular transaction T. In this subsection, we define a notion of "visibility" of one transaction to another. This notion is a technical one, but one that is natural and convenient in the formal statements of results and in their proofs. Visibility is defined so that, in the usual transaction-processing systems, only a transaction T' that is visible to another transaction T can affect the behavior of T.

A transaction T' can affect another transaction T in several ways. First, if T' is an ancestor of T, then T' can affect T by passing information down the transaction tree via invocations. Second, a transaction T' that is not an ancestor of T can affect T through COMMIT actions for T' and all ancestors of T' up to the level of the least common ancestor with T; information can be propagated from T' up to the least common ancestor via COMMIT actions, and from there down to T via invocations. Third, a transaction T' that is not an ancestor of T can affect T by accessing an object that is later accessed by T; in most of the usual transaction-processing algorithms, this is only allowed to occur if there are intervening COMMIT actions for all ancestors of T' up to the level of the least common ancestor with T.

Thus, we define "visibility" as follows. Let β be any sequence of serial actions. If T and T' are transaction names, we say that T' is *visible* to T in β if there is a COMMIT(U) action in β for every U in ancestors(T') - ancestors(T). Thus, every ancestor of T' up to (but not necessarily including) the least common ancestor of T and T' has committed in β.

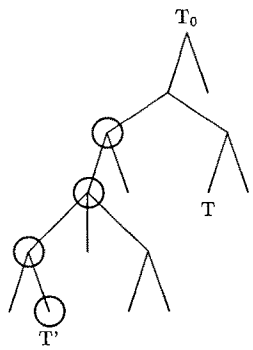

Figure 5: Visibility

Figure 5 depicts two transactions, T and T' neither an ancestor of the other. If the transactions represented by all of the circled nodes have committed in some sequence of serial actions, then the definition implies that T' is visible to T.

The following lemma describes elementary properties of "visibility".

Lemma 19: Let β be a sequence of actions, and let T, T' and T'' be transaction names.

1. If T' is an ancestor of T, then T' is visible to T in β.
2. T' is visible to T in β if and only if T' is visible to lca(T,T') in β.
3. If T'' is visible to T' in β and T' is visible to T in β, then T'' is visible to T in β.
4. If T' is live in β and T' is visible to T in β, then T is a descendant of T'.
5. If T' is an orphan in β and T' is visible to T in β, then T is an orphan in β.

We use the notion of "visibility" to pick, out of a sequence of actions, a subsequence consisting of the actions corresponding to transactions that are visible to a given transaction T. More precisely, if β is any sequence of actions and T is a transaction name, then *visible*(β,*T*) denotes the subsequence of β consisting of serial actions π with hightransaction(π) visible to T in β. Note that every action occurring in visible(β,T) is a serial action, even if β itself contains other actions. The following obvious lemma says that the "visible" operator on sequences picks out either all or none of the actions having a particular transaction.

Lemma 20: Let β be a sequence of actions, and let T and T' be transaction names. Then visible(β,T)|T' is equal to β|T' if T' is visible to T in β, and is equal to the empty sequence otherwise.

5.2 Event and Transaction Orders

The hypothesis of the theorem refers to rearranging the operations at each object according to a given order on transactions. The definitions required to describe the appropriate kind of ordering to use for this purpose are provided in this subsection.

5.2.1 Affects Order

We first define a partial order "affects(β)" on the events of a sequence β of serial actions. This will be used to describe basic dependencies between events in a simple behavior; any appropriate ordering will be required to be consistent with these dependencies. We define the affects relation by first defining a subrelation that we call the "directly-affects" relation and then taking the transitive closure. For a sequence β of serial actions, and events ϕ and π in β, we say that ϕ *directly affects* π in β (and that $(\phi,\pi) \in$ directly-affects(β)) if at least one of the following is true.

- transaction(ϕ) = transaction(π) and ϕ precedes π in β,[9]
- ϕ = REQUEST_CREATE(T) and π = CREATE(T)
- ϕ = REQUEST_COMMIT(T,v) and π = COMMIT(T)
- ϕ = REQUEST_CREATE(T) and π = ABORT(T)
- ϕ = COMMIT(T) and π = REPORT_COMMIT(T,v)
- ϕ = ABORT(T) and π = REPORT_ABORT(T)

If β is a simple behavior, and $(\phi,\pi) \in$ directly-affects(β), then it is easy to see that ϕ precedes π in β. For a sequence β of serial events, define the relation affects(β) to be the transitive closure of the relation directly-affects(β). If the pair (ϕ,π) is in the relation affects(β), we also say that ϕ *affects* π in β. The following is immediate.

---

[9]This includes accesses as well as non-accesses.

Lemma 21: Let β be a simple behavior. Then affects(β) is an irreflexive partial order on the events in β.[10]

The conditions listed in the definition of "directly-affects" should seem like a reasonable collection of dependencies among the events in a simple behavior. Here we try to give some technical justification for these conditions. In the proof of the theorem, we will attempt to extract serial behaviors from a given simple behavior. The transaction orderings used to help in this construction will be constrained to be consistent with "affects"; this will mean that the sequences we construct will be closed under "affects" and that the orders of events in these sequences are consistent with "affects". Thus, if β is a simple behavior and $(\phi,\pi) \in$ directly-affects(β), all the serial behaviors we construct that contain π will also contain ϕ, and ϕ will precede π in each such behavior.

The first case of the "directly-affects" definition is used because we are not assuming special knowledge of transaction behavior; if we included π and not ϕ in our candidate serial behavior, we would have no way of proving that the result included correct behaviors of the transaction automata. The remaining cases naturally parallel the preconditions of the serial scheduler; in each case, the preconditions of π as an action of the serial scheduler include a test for a previous occurrence of ϕ, so a sequence of actions with π not preceded by ϕ could not possibly be a serial behavior.

As before, we extend the "affects" definition to sequences β of arbitrary actions by saying that ϕ affects π in β exactly if ϕ affects π in serial(β).

5.2.2 Sibling Orders

The type of transaction ordering needed for our theorem is more complicated than that used in the classical theory, because of the nesting involved here. Instead of just arbitrary total orderings on transactions, we will use orderings that only relate siblings in the transaction nesting tree. We call such an ordering a "sibling order". Interesting examples of sibling orders are the order of completion of transactions or an order determined by assigned timestamps.

Let *SIB* be the (irreflexive) sibling relation among transaction names, for a particular system type; thus, $(T,T') \in$ SIB if and only if $T \neq T'$ and parent(T) = parent(T'). If $R \subseteq$ SIB is an irreflexive partial order then we call R a *sibling order*. Sibling orders are the analog for nested transaction systems of serialization orders in single-level transaction systems. Note that sibling orders are not necessarily total, in general; totality is not always appropriate for our results.

A sibling order R can be extended in two natural ways. First, R_{trans} is the extension of R to descendants of siblings, i.e., the binary relation on transaction names containing (T,T') exactly when there exist transaction names U and U' such that T and T' are descendants of U and U' respectively, and $(U,U') \in R$. This order echoes the manner in which the serial scheduler runs transactions when it runs siblings with no concurrency, in the order specified by R. Second, if β is any sequence of actions, then $R_{event}(\beta)$ is the extension of R to serial events in β, i.e., the binary relation on events in β containing (ϕ,π) exactly when ϕ and π are distinct serial events in β with lowtransactions T and T' respectively, where $(T,T') \in R_{trans}$. It is easy to see that R_{trans} is an irreflexive partial order, and for any sequence β of actions, $R_{event}(\beta)$ is an irreflexive partial order.

The concept of a "suitable sibling order" describes two basic conditions that will be required of the sibling orders to be used in our theorem. The first condition is a technical one asserting that R orders sufficiently many siblings, while the second condition asserts that R does not contradict the dependencies described by the affects relation. Let β be a sequence of actions and T a transaction name. A sibling order R is *suitable* for β and T if the following conditions are met.

1. R orders all pairs of siblings T' and T'' that are lowtransactions of actions in visible(β,T).

---

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

2. $R_{event}(\beta)$ and affects(β) are consistent partial orders on the events in visible(β,T).[11]

5.3 The Serializability Theorem

We now present the main result. It says that a simple behavior β is serially correct for a non-orphan transaction name T provided that there is a suitable sibling order R for which a certain "view condition" holds for each object name X. The view condition says that the portion of β occurring at X that is visible to T, reordered according to R, is a behavior of the serial object S(X). In order to make all of this precise, suppose β is a finite simple behavior, T a transaction name, R a sibling order that is suitable for β and T, and X an object name. Let ξ be the sequence consisting of those operations occurring in β whose transaction components are accesses to X and that are visible to T in β, ordered according to R_{trans} on the transaction components. (The first condition in the definition of suitability implies that this ordering is uniquely determined.) Define *view*(β,*T*,*R*,*X*) to be perform(ξ).

Thus, view(β,T,R,X) represents the portion of the behavior β occurring at X that is visible to T, reordered according to R. Stated in other words, this definition extracts from β exactly the REQUEST_COMMIT actions for accesses to X that are visible to T; it then reorders those REQUEST_COMMIT actions according to R, and then inserts an appropriate CREATE action just prior to each REQUEST_COMMIT action. The theorem uses a hypothesis that each view(β,T,R,X) is a behavior of the serial object S(X) to conclude that β is serially correct for T.

Theorem 22: (Serializability Theorem) Let β be a finite simple behavior, T a transaction name such that T is not an orphan in β, and R a sibling order suitable for β and T. Suppose that for each object name X, view(β,T,R,X) ∈ finbehs(S(X)). Then β is serially correct for T.

Proof: Given β, T and R, the needed serial behavior is constructed explicitly. The construction is done in several steps. First, visible(β,T), the portion of β visible to T, is extracted from β. This sequence is then reordered according to R and affects(β). (There may be many ways of doing this.) The reordered sequence is then truncated at an appropriate place, just after the last action involving T or any of its descendants. The resulting sequence γ is shown to be a serial behavior by showing separately that its projections are behaviors of the transaction automata, of the serial object automata, and of the serial scheduler, and then applying Proposition 4.

If T' is a nonaccess transaction name, Proposition 1 implies that β|T' is a behavior of A(T'). Proposition 20 and the fact that $R_{event}(\beta)$ is consistent with affects(β) ensure that γ|T' is a prefix of β|T' and so is a behavior of A(T'). Thus, the projection of γ on each of the transaction automata is a behavior of that automaton.

For each object name X, unwinding the definitions shows that γ|X is a prefix of view(β,T,R,X). The "view condition" hypothesis of the theorem, that view(β,T,R,X) ∈ finbehs(S(X)), implies that γ|X is a behavior of S(X). Thus, the projection of γ on each of the serial object automata is a behavior of that automaton.

Finally, an explicit argument by induction on the length of γ shows that γ is a behavior of the serial scheduler automaton. Consistency with affects(β) is used to show that certain events are included in γ; this implies that the serial scheduler preconditions involving occurrence of certain events are satisfied. The properties of the "visible" operator are used to show that certain events, e.g., those involving live transactions neither ancestors nor descendants of T, are not included in γ; this implies that the serial scheduler preconditions involving nonoccurrence of certain actions are satisfied.

The theorem has a straightforward corollary that outlines a strategy for showing that a particular system satisfies the correctness condition in which we are mainly interested, i.e., that all its finite behaviors are serially correct for T_0.

---

[11]Two binary relations R and S are *consistent* if their union can be extended to an irreflexive partial order (or in other words, if their union has no cycles).

Corollary 23: Let $\{B_i\}_{i \in I}$ be a strongly compatible set of automata and let $B = \Pi_{i \in I} B_i$. Suppose that the name T_0 is in the index set I, and that the automaton $A(T_0)$ is associated with T_0 in B. Suppose that for every finite behavior β of B, the following conditions hold.

1. serial(β) is a simple behavior.
2. There exists a sibling order R suitable for serial(β) and T_0, such that for each object name X, view(serial(β),T_0,R,X) $\in$ finbehs(S(X)).

Then every finite behavior of B is serially correct for T_0.

6 Applications of the Serializability Theorem

We use this theorem elsewhere in our work to reason about the correctness of a wide variety of algorithms for implementing atomic transactions. In particular, we carry out correctness proofs for several algorithms that use locking and others that use timestamps.

The locking algorithm of Moss [Moss] is designed for data objects that are accessible only by read and write operations. We have developed a similar algorithm, in [FLMW2], that accomodates arbitrary data types. These algorithms involve simultaneous locking at different levels of the transaction nesting tree. A transaction is only permitted to access a data object if it has a suitable lock on that object. Sometime after a transaction commits, its locks are passed up to its parent and associated modifications to the data are made available to the parent and its other descendants. On the other hand, when a transaction aborts, its locks are released and its modifications to the data are discarded. The decision about whether to permit an access transaction to obtain a lock is based on whether any locks for "conflicting" operations are held by transactions that are not ancestors of the given access.

Using Corollary 23 above, we can prove that all the finite behaviors of a system B are serially correct for T_0 if B uses these algorithms. Although the locking algorithms include more actions than the simple system, it is not hard to see that serial(β) is a simple behavior, for every finite behavior β of B. The sibling order R used in the proof is the "completion order", i.e., the order in which sibling transactions commit and abort. Proving correctness of this algorithm using the Serializability Theorem highlights the key reason why locking algorithms work: roughly speaking, the condition that view(β,T_0,R,X) $\in$ finbehs(S(X)) says that the processing at any object is "consistent" with the transaction completion order. The "consistency" mentioned here means that reordering the appropriate, "visible" portion of the processing at each object in completion order yields a correct behavior for the corresponding serial object automaton. We can also use the Serializability Theorem to prove the stronger statement that the locking algorithms mentioned above are serially correct for all non-orphan transactions.

Our correctness proofs for these algorithms have an interesting structure. Namely, we describe each algorithm as the composition of a component automaton for each object plus one global "controller" automaton that simply manages communication among the other automata. A local condition called "dynamic atomicity" is defined; this condition essentially says that the object satisfies the view condition using the completion order. The Serializability Theorem implies that if all the objects are dynamic atomic, the system guarantees serial correctness for all non-orphan transaction names. The rest of the proof involves showing that the objects that model the given locking algorithms are dynamic atomic.

This proof structure allows us to obtain much stronger results than just the correctness of the given algorithms. As long as each object is dynamic atomic, the whole system will guarantee that any finite behavior is serially correct for all non-orphan transaction names. Thus, we are free to use an arbitrary implementation for each object, independent of the choice of implementation for each other object, as long as dynamic atomicity is satisfied. For example, a simple algorithm such as Moss' can be used for most objects, while a more sophisticated algorithm permitting extra concurrency by using type-specific information can be used for objects that are "hot spots" (that is, very frequently accessed.) The idea of a local condition that guarantees serial correctness was introduced by Weihl [Weihl] for systems without transaction nesting.

The timestamp algorithm of Reed [Reed] is designed for data objects that are accessible only by read

and write operations. We have developed a similar algorithm, in [AFLMW], that accomodates arbitrary data types. (This work generalizes work of Herlihy [Herlihy] giving a timestamp algorithm for single-level transactions using arbitrary data types.) These algorithms both involve assignment of ranges of timestamp values to transactions in such a way that the interval of a child transaction is included in the interval of its parent, and the intervals of siblings are disjoint. Responses to accesses are determined from previous accesses with earlier timestamps.

We again analyze these algorithms using the Serializability Theorem and its Corollary. This time, the sibling order used is the timestamp order. Now the condition that view(β,T_0,R,X) $\in$ finbehs(S(X)) says that the processing of accesses to X is "consistent" with the timestamp order, in that reordering the processing in timestamp order yields a correct behavior for the corresponding serial object automaton. The Corollary then implies that all finite behaviors are serially correct for T_0, and the Serializability Theorem implies that the timestamp algorithms are serially correct for all non-orphan transaction names. Once again, each algorithm is described as the composition of object automata and a controller. This time, a local condition called "static atomicity" is used, saying that an object satisfies the view condition using the timestamp order. As long as each object is static atomic, the whole system is serially correct for non-orphan transactions. We show that both Reed's algorithm and our version of Herlihy's algorithm ensure static atomicity. Again, we have the flexibility to implement objects independently as long as static atomicity is guaranteed.

Objects can be proved to be dynamic atomic or static atomic using standard assertional proof techniques and connections between the object's state and history. It is also possible to prove that some objects are dynamic atomic or static atomic by showing that they implement other objects of the same kind. Possibilities maps and Proposition 6 can be used to show this. This strategy is especially useful in cases where the object keeps information in a compact form, whereas the required local property is easy to prove for a less compact variant of the algorithm. We refer the interested reader to [FLMW2] and [AFLMW] for more details.

7 Conclusions

In this paper, we have presented correctness conditions for atomic transaction systems. These conditions are stated at the user interface to the system, which is the interface of primary interest. The fact that the conditions are stated at this interface makes them quite general; they can be used to state appropriate correctness conditions for a wide variety of different algorithms. We have also described one general theorem, the Serializability Theorem, which is useful for proving correctness of many interesting and apparently dissimilar algorithms.

The Serializability Theorem is not the only tool we use for our correctness proofs. There are several other techniques that we use for decomposing proofs of transaction-processing algorithms. For example, in [GL], we provide proofs for replicated data algorithms based on the quorum consensus technique of Gifford [Gifford]. We consider replication management algorithms in combination with concurrency control and recovery algorithms. Our presentation separates the concerns very cleanly: the algorithm is divided into modules that handle replication and modules that handle the concurrency control and recovery. Correctness conditions for the two separate algorithms are combined to yield correctness for the complete algorithm. In particular, all that is required of the concurrency control and recovery algorithms is that they guarantee serial correctness for non-orphan transactions (with respect to the individual copies of the data objects); thus, there is considerable flexibility in the choice of concurrency control and recovery algorithms. We remark that transaction nesting provides a particularly good way to organize this decomposition: the replication part of the algorithm is formally described in terms of new copy-management subtransactions that are called by the user-level transactions in place of the original user-level accesses to objects.

We expect that our model in general, and the Serializability Theorem in particular, will prove quite useful for reasoning about many more algorithms than those that we have already considered. We are also particularly interested in understanding how to reason about multi-level locking algorithms such as that considered in [BBG], and in understanding complicated algorithms that are used for concurrency

control and recovery in an environment having volatile memory which is lost during a crash. Understanding these ideas in our model is work remaining to be done.

Finally, we remark that our Serializability Theorem still seems somewhat more complicated than the classical theorem, even taking the generalizations into account. The classical theorem was stated in simple combinatorial terms, while our theorem involves a more complicated fine-grained treatment of individual actions. We wonder if it is possible to combine the advantages of the two approaches: perhaps there is a simple combinatorial condition that takes suitable account of nesting and failures, and that implies the natural and general correctness conditions described in this paper.

8 References

[AFLMW] Aspnes, J., Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "A Theory of Timestamp-Based Concurrency Control for Nested Transactions," Proceedings of 14th International Conference on Very Large Data Bases, to appear.

[BBG] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Technical Report, Wang Institute TR-86-03, March 1986.

[BHG] Bernstein, P., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.

[Davies] Davies, C. T., "Recovery Semantics for a DB/DC System," Proceedings of 28th ACM National Conference, 1973, pp. 136-141.

[FLMW1] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "Nested Transactions and Read/Write Locking," Proceedings of 6th ACM Symposium on Principles of Database Systems, 1987, pp. 97-111. An expanded version is available as Technical Memo MIT/LCS/TM-324, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., April 1987.

[FLMW2] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "Nested Transactions, Conflict-Based Locking and Dynamic Atomicity," Technical Memo MIT/LCS/TM-340, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., September 1987. Submitted for publication.

[Gifford] Gifford, D., "Weighted Voting for Replicated Data," Proceedings of 7th ACM Symosium on Operating System Principles, 1979, pp. 150-162.

[GL] Goldman, K., and Lynch, N., "Nested Transactions and Quorum Consensus," Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987, pp. 27-41. An expanded version is available as Technical Report MIT/LCS/TR-390, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., May 1987.

[Herlihy] Herlihy, M., "Extending Multiversion Time-Stamping Protocols to Exploit Type Information," IEEE Transactions on Computers C-36, April 1987.

[HLMW] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W., "On the Correctness of Orphan Elimination Algorithms," Proceedings of 17th IEEE Symposium on Fault-Tolerant Computing, 1987, pp. 8-13.

[Hoare] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall International, 1985.

[Liskov] Liskov, B., "Distributed Computing in Argus," Communications of ACM, vol. 31, no. 3, March 1988, pp. 300-312.

[LM] Lynch, N., and Merritt, M., "Introduction to the Theory of Nested Transactions," Technical Report MIT/LCS/TR-367, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., July 1986. To appear in Theoretical Compter Science.

[LMWF] Lynch, N., Merritt, M., Weihl, W., and Fekete, A., "Atomic Transactions," in preparation.

[LT] Lynch, N., and Tuttle, M., "Hierarchical Correctness Proofs for Distributed Algorithms," Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987, pp. 137-151. An expanded version is available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., April 1987.

[Milner] Milner, R., "A Calculus of Communicating Systems," Lecture Notes in Computer Science 92, Springer Verlag, 1980.

[Moss] Moss, J. E. B., "Nested Transactions: An Approach To Reliable Distributed Computing," Ph.D. Thesis, Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., April 1981. Also, published by MIT Press, March 1985.

[Perl] Perl, S., "Distributed Commit Protocols for Nested Atomic Actions," M.S. Thesis, M.I.T. September 1987.

[SS] Spector, A., and Swedlow, K. (eds), "Guide to the Camelot Distributed Transaction Facility: Release 1," Carnegie Mellon University, Pittsburgh, PA., October 1987.

[Reed] Reed, D. P., "Naming and Synchronization in a Decentralized Computer System," Ph.D Thesis, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., September 1978.

[Weihl] Weihl, W., "Specification and Implementation of Atomic Data Types," Ph.D. Thesis, Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., March 1984.