Report Number: WUCS-93-35

1993-01-01

A Unified Model for Shared-Memory and Message-Passing Systems

Kenneth Goldman and Katherine Yelick

A unified model of distributed systems that accomodates both shared-memory and message-passing communication is proposed. An extension of the I/O automaton model of Lynch and Tuttle, the model provides a full range of types of atomic accesses to shared memory, from basic reads and writes to read-modify-write. In addition to supporting the specification and verification of shared memory algorithms, the unified model is particularly helpful for proving correspondences between atomic shared objects and invocation-response systems and for proving the correctness of systems that contain both message passing and shared memory (such as a network of shared-memory multiprocessors or a... **Read complete abstract on page 2.**

A Unified Model for Shared-Memory and Message-Passing Systems

Kenneth Goldman and Katherine Yelick

Complete Abstract:

A unified model of distributed systems that accomodates both shared-memory and message-passing communication is proposed. An extension of the I/O automaton model of Lynch and Tuttle, the model provides a full range of types of atomic accesses to shared memory, from basic reads and writes to read-modify-write. In addition to supporting the specification and verification of shared memory algorithms, the unified model is particularly helpful for proving correspondences between atomic shared objects and invocation-response systems and for proving the correctness of systems that contain both message passing and shared memory (such as a network of shared-memory multiprocessors or a distributed memory multiprocessor with multi-threaded nodes). As an illustration of the model, we consider distributed systems in which the shared objects have the linearizability property proposed by Herlihy and Wing. We use the model to construct a careful proof that invocation-response systems constructed from linearizable objects simulate atomic shared memory systems. In addition, we extend the work of Herlihy and Wing by treating not only safety properties of invocation-response systems, but also liveness properties.

A Unified Model for Shared-Memory and Message-Passing Systems

Kenneth Goldman and Katherine Yelick

WUCS-93-35

June 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899

A Unified Model for Shared-Memory and Message-Passing Systems

Kenneth Goldman[*]
Department of Computer Science
Washington University
St. Louis, MO 63130
kjg@cs.wustl.edu

Katherine Yelick[†]
Computer Science Division
University of California
Berkeley, CA 94720
yelick@cs.berkeley.edu

June 29, 1993

Abstract

A unified model of distributed systems that accommodates both shared-memory and message-passing communication is proposed. An extension of the I/O automaton model of Lynch and Tuttle, the model provides a full range of types of atomic accesses to shared memory, from basic reads and writes to read-modify-write. In addition to supporting the specification and verification of shared memory algorithms, the unified model is particularly helpful for proving correspondences between atomic shared objects and invocation-response systems and for proving the correctness of systems that contain both message passing and shared memory (such as a network of shared-memory multiprocessors or a distributed memory multiprocessor with multi-threaded nodes). As an illustration of the model, we consider distributed systems in which the shared objects have the *linearizability* property proposed by Herlihy and Wing. We use the model to construct a careful proof that invocation-response systems constructed from linearizable objects simulate atomic shared memory systems. In addition, we extend the work of Herlihy and Wing by treating not only safety properties of invocation-response systems, but also liveness properties.

Keywords: distributed systems, I/O automata, invocation-response, linearizability, message passing, models, shared memory

1 Introduction

Reasoning about algorithms for asynchronous concurrent systems is difficult, primarily because of the arbitrary interleaving of process steps that may occur in an execution. As a result, researchers have turned to formal models in order to define problems precisely, give unambiguous descriptions

[*]This work was supported in part by the National Science Foundation as a Research Initiation Award (number CCR-91-10029). A portion of this work has appeared as: Kenneth Goldman and Nancy Lynch, Modelling Shared State in a Shared Action Model, *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990, and was supported in part by the National Science Foundation under Grant CCR-86-11442, by the Office of Naval Research under Contract N00014-85-K-0168, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

[†]This work was supported in part by the National Science Foundation as a Research Initiation Award (number CCR-9210260), and as an Infrastructure Grant (number CDA-8722788), by Lawrence Livermore National Laboratory (task number 33), by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Semiconductor Research Consortium under contract 92-DC-008 and 93-DC-008, and by AT&T. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

of algorithms, and construct careful proofs for safety and progress properties. These models allow one to be explicit about the possible interleavings that may occur in a distributed system and may specify which of those interleavings are to be considered "fair" to the individual system components. Examples include CSP [9], in which system components communicate by sending messages over synchronous channels, and UNITY [4], in which components communicate by reading and modifying shared variables.

The I/O automaton model [15, 16] is particularly well-suited for modelling distributed algorithms described using message passing. The I/O automaton model is a (not necessarily finite) state machine model that provides extra support for classifying actions as input or output and for describing fairness conditions. Precise problem statements are defined in terms of the input and output actions that occur at the boundary between the algorithm and its "environment." These problem statements may include nontrivial liveness constraints on the behavior of the algorithm. Careful algorithm descriptions are constructed by specifying the states and transition relations of I/O automata. A range of proof techniques, from simple assertional reasoning to hierarchical possibilities mappings, may be used to verify an algorithm satisfies a problem statement. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results. The communication mechanism in a distributed system is modeled as an explicit I/O automaton that shares actions with the other system components. Therefore, the model can accommodate a variety of message-passing systems, from systems with strictly FIFO message delivery to those in which messages may be delivered out of order or not at all.

Although the I/O automaton model provides excellent support for modelling message-passing algorithms, many of the important asynchronous concurrent algorithms are described using shared memory. Also, one might wish to use both shared-memory and message-passing to describe different components of a heterogeneous system. Therefore, introducing a shared-memory mechanism into the I/O automaton model is a useful unification of these two approaches. The shared memory model of Lynch and Fischer [13] introduced the separation of input and output actions, and was a precursor of the current I/O automaton model. However, until now it has not been clear how to integrate the two basic approaches.

In this paper, we present an extension to the I/O automaton model to allow modelling of shared memory systems, as well as systems that have both shared memory and shared action communication. A full range of types of atomic accesses to shared memory is allowed, from basic reads and writes to atomic read-modify-write. We define a special class of actions, called "shared memory actions," to model atomic accesses to shared memory. Actions in the basic I/O Automaton model contain information about input and output values; we define shared memory actions to contain additional information that corresponds to the contents of the shared memory before and after the action occurs. A "shared memory automaton" is then defined to be an I/O automaton that satisfies certain natural conditions regarding its shared memory actions. For example, one condition captures the idea that an access to shared memory must be prepared to observe any value in the memory.

Since shared memory automata are simply special cases of I/O automata, all the I/O automaton model definitions and properties, notably composition and fairness, apply to shared memory automata as well. We show that composing of a collection of shared memory automata (for a given set of shared variables) yields another shared memory automaton (for the same set of variables). To combine shared memory automata having different (not necessarily disjoint) sets of shared variables, we define an "augmentation" operator that is used to expand the set of shared variables for each component before composing. We show that the natural compositionality results hold when we combine shared memory automata in this way. For example, projecting the execution of a composition on the individual components yields executions of those components. Since we expose the observed state of shared memory in the behavior of an automaton, we also achieve compositionality of the *behaviors* of shared memory automata. That is, in the standard sense of I/O automaton composition, the behaviors of a composition of shared memory automata are the same as the composition of the behaviors of the individual automata.

Shared memory automata operate in a system in which the environment is free to change the contents of the shared memory at any time. We define a "closeout" operator, which takes a shared memory automaton and a set of variables and produces a new shared memory automaton in which the given set of variables is made private, absorbed into the local state. In this way, we restrict the set of components in a system that may access portions of the shared memory.[1] We provide an analogous closeout operator on sets of behaviors, and we show that the behaviors of a *closed out automaton* are the same as the *closed out behaviors* of the original automaton.

Just as does the original I/O automaton model, our extended model supports careful problem specification (including both safety and progress properties), unambiguous system description, verification and analysis. Both safety and progress properties of algorithms may be shown using standard proof techniques (e.g., invariant assertions and variant functions). For example, these techniques have been used within the extended model for proving the correctness of Dijkstra's classical shared memory mutual exclusion algorithm [7].

Although the shared memory extensions do expand the capabilities of the I/O automaton model to include support for shared memory algorithms, the major benefit of a unified model that supports both atomic shared memory and message passing is not merely the ability to reason about shared memory algorithms and message passing algorithms with the same set of tools. The major benefit is the ability to combine shared memory and message passing in a single formal setting. For example, the extended model has been used as the basis of a new methodology for reasoning about recursive distributed algorithms in which each recursive invocation of an algorithm is modeled as a separate automaton using dynamic process creation. The various recursive instantiations of the algorithm on a given processor communicate with their corresponding instantiations on other processors through message passing, and they communicate among themselves through shared memory [17].

In addition, a unified model provides the ability to describe systems that use both shared memory and message passing communication. For example, systems like the Alewife multiprocessor [1] or the Split-C language [5] implement a global shared memory abstraction using a combination of message passing between processors and shared memory accesses between threads (or handlers) on a single processor. Reasoning about the behavior of these systems requires a mixture of shared memory and message passing semantics. Another class of systems that mix message passing and shared memory are programs (typically scientific applications) implemented on a heterogeneous network of workstations and multiprocessors. Within a multiprocessor, communication may be done using shared memory, while communicating across the network involves message passing. Numerous examples of systems exist to support such heterogeneous network computing, with one of the most popular being PVM [6].

Perhaps one of the most important benefits of a unified model is that of providing a context in which to understand and prove correspondences between shared memory and message passing systems. We emphasize that both shared memory and message passing are modelled *directly*; in no sense is one "implemented" on top of the other. Therefore, one may describe both kinds of systems using the primitives of the model, and then formally relate the two.

To illustrate this benefit of our unified model, we consider the problem of simulating an atomic shared memory in a distributed system where message passing is the physical means of communica-

[1]The ability to closeout with respect to a subset of the shared variables (as opposed to the entire set) may be likened to lexical scoping of variable declarations in a conventional programming language.

tion. The motivation for simulating an atomic shared memory system on a message passing system is that algorithms described using atomic accesses to shared memory are easy to reason about, since one is not concerned with the possible interleavings of invocations and responses for object access. However, the invocation-response approach, in which the invocation of an operation on an object and the corresponding response are modelled as separate atomic steps, fits more naturally with multiprocessor architectures and often supports greater concurrency. Thus, an important problem in multiprocessor algorithm design is to carefully construct the processes and objects in such a way that any the invocation-response system "simulates" an atomic access system. One approach to this problem has been advanced by Herlihy and Wing [8]. They propose a property of objects, called *linearizability*, that permits one to construct invocation-response systems, and then to reason about only those executions in which each response immediately follows the corresponding invocation.

We exercise the unified model resulting from our shared memory extensions by proving a useful relationship between the invocation-response approach and the atomic access approach in the context of linearizable objects. In addition, we extend the work of Herlihy and Wing by treating not only safety properties of invocation-response systems, but liveness properties as well. The proof rests on a simulation argument, namely that systems of linearizable objects "simulate" atomic shared memory systems. The unified model assists us in this proof possible because we are able to express both the invocation-response system and the atomic shared memory system within the same model, and thereby are able to relate the two.

The remainder of the paper is organized as follows. In Section 2, we review the I/O automaton model. We define our extensions for shared memory in Section 3 and show some important properties that follow from these definitions. Next, in Section 4, we use the extended model to establish a formal relationship between invocation-response systems and atomic access systems. The paper concludes with a summary and discussion.

2 The I/O Automaton Model

The I/O automaton model of Lynch and Tuttle [15, 16] is the starting point for this work. One of the I/O automaton model's distinguishing features is the clear separation of input and output actions. This will be particularly important for our shared memory extensions for two reasons. First, the fact that each action is under the control of exactly one component means that simply by using output actions to model updates to the shared memory, we capture the notion of a single module making an atomic update to shared memory (without any active participation by other modules). Second, the fact that input actions are always enabled means that we can write modules that passively observe the shared memory accesses by others without interfering. Other features of the model important to us are its treatment of fairness and its compositionality properties.

Before describing our extensions to the I/O automaton model, we present a brief introduction to the I/O automaton model. This brief introduction is adapted from [16], which explains the model in more detail, presents examples, and includes comparisons to other models.

2.1 I/O Automata

I/O automata are best suited for modelling systems in which the components operate asynchronously. Each system component is modeled as an I/O automaton, which is essentially a nondeterministic (possibly infinite state) automaton with an action labeling each transition. An automaton's actions are classified as either 'input', 'output', or 'internal'. 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. An automaton is said to be *closed* if it has no input actions; it models a closed system that does not interact with its environment.

Formally, an *action signature* S is a partition of a set $acts(S)$ of *actions* into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of *input actions*, *output actions*, and *internal actions*, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of *external actions.* We denote by $local(S) = out(S) \cup int(S)$ the set of *locally-controlled actions.* An I/O automaton consists of five components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
- an equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

The equivalence relation $part(A)$ will be used in the definition of fair computation. Each class of the partition may be thought of as a separate process. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. 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.* This means that the automaton is unable to block its input.

An *execution* of A is a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i and $s_0 \in start(A)$. The *schedule* of an execution α is the subsequence of α consisting of the actions appearing in α. The *behavior* of an execution or schedule α of A is the subsequence of α consisting of *external* actions. The sets of executions, finite executions, schedules, finite schedules, behaviors, and finite behaviors are denoted $execs(A)$, $finexecs(A)$, $scheds(A)$, $finscheds(A)$, $behs(A)$, and $finbehs(A)$, respectively. The same action may occur several times in an execution or a schedule; we refer to a particular occurrence of an action as an *event.*

The separation of input and output actions will be important in our shared memory extensions for two reasons. First, the fact that each action is under the control of exactly one component means that by simply using actions to model updates to the shared memory, we capture the notion of a single module making an atomic update to shared memory (without any active participation by other modules). Second, the fact that input actions are always enabled means that we can use shared memory input actions to construct modules that passively observe the shared memory accesses by others without interfering.

2.2 Composition

We can construct an automaton modelling a complex system by composing automata modelling the simpler system components. When we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π , all automata having π as an action perform π simultaneously (automata not having π as an action do nothing).

Since we require that at most one system component controls the performance of any given action, we must place some compatibility restrictions on the collections of automata that may be composed. A countable collection $\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible* if for all $i, j \in I$ satisfying $i \neq j$ 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 contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata are *strongly compatible* if their action signatures are strongly compatible.

The *composition* $S = \prod_{i \in I} S_i$ of a countable 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
- $int(S) = \cup_{i \in I} int(S_i)$.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:[2]

- $sig(A) = \prod_{i \in I} sig(A_i)$,
- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $steps(A)$ is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in steps(A_i)$, and if $\pi \notin acts(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$, and
- $part(A) = \cup_{i \in I} part(A_i)$.

Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1} \ldots$ of A, let $\alpha|A_i$ (read "α projected on A_i") be the sequence obtained by deleting $\pi_j\vec{s_j}$ when $\pi_j \notin acts(A_i)$ and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

2.3 Fairness

Of all the executions of an I/O automaton, we are primarily interested in the 'fair' executions — those that permit each of the automaton's primitive components (i.e., its classes or processes) to have infinitely many chances to perform output or internal actions. The definition of automaton composition says that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the essential structure of the system's primitive components. In the model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of *part*(A):

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

[2]Here $start(A)$ and $states(A)$ are defined in terms of the ordinary Cartesian product, while $sig(A)$ is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$.

We denote the set of fair executions of A by *fairexecs*(A). We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by *fairbehs*(A). Similarly, β is a *fair schedule* of A if β is the schedule of a fair execution of A, and we denote the set of fair schedules of A by *fairscheds*(A).

The definitions of composition and fairness imply certain natural relationships between the (fair) executions of a composition and the (fair) executions of the individual components. For example, the following lemma from [16] states that (fair) executions of component automata can often be pasted together to form a (fair) execution of the composition.

Lemma 1: Let $\{A_i\}_{i \in \mathcal{I}}$ be a strongly compatible collection of automata and let $A = \Pi_{i \in \mathcal{I}} A_i$. Suppose α_i is a (fair) execution of A_i for every $i \in I$, and suppose β is a sequence of actions in *acts*(A) such that $\beta | A_i = \mathit{sched}(\alpha_i)$ for every $i \in \mathcal{I}$. Then there is an (fair) execution α of A such that $\beta = \mathit{sched}(\alpha)$ and $\alpha_i = \alpha | A_i$ for every $i \in \mathcal{I}$. Moreover, the same result holds when *acts* and *sched* are replaced by *ext* and *beh*, respectively.

2.4 Problem Specification

A 'problem' to be solved by an I/O automaton is formalized as a set of (finite and infinite) sequences of external actions. An automaton is said to *solve* a problem P provided that its set of fair behaviors is a subset of P. Although the model does not allow an automaton to block its environment or eliminate undesirable inputs, we can formulate our problems (i.e., correctness conditions) to require that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs.

We want a problem specification to be an interface together with a set of behaviors. We therefore define a *schedule module* H to consist of two components, an action signature *sig*(H), and a set *scheds*(H) of *schedules*. Each schedule in *scheds*(H) is a finite or infinite sequence of actions of H. Subject to the same restrictions as automata, schedule modules may be composed to form other schedule modules. The resulting signature is defined as for automata, and the schedules *scheds*(H) is the set of sequences β of actions of H such that for every module H' in the composition, $\beta | H'$ is a schedule of H'.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors. A set of sequences $\mathcal{P}$ is said to be *prefix-closed* if $\beta \in \mathcal{P}$ whenever both β is a prefix of α and $\alpha \in \mathcal{P}$. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that *finbehs*(M) is prefix-closed. Let M be a prefix-closed module and let $\mathcal{P}$ be a nonempty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap \mathit{int}(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi | \Phi \in \mathcal{P}$ whenever $\beta | \Phi \in \mathcal{P}$, $\pi \in \mathit{out}(M)$, and $\beta\pi | M \in \mathit{finbehs}(M)$. Informally, a module *preserves* a property $\mathcal{P}$ iff the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. One can prove that a composition preserves a property by showing that each of the component automata preserves the property.

3 Shared Memory Definitions

In this section, we present a set of definitions that extends the I/O automaton model in order to allow modelling shared memory algorithms. *We do not redefine any concepts,* but simply add new concepts to the existing model. We model each system component that accesses shared memory as a restricted I/O automaton called a "shared memory automaton". The fact that shared memory automata are simply special cases of I/O automata means that all the standard definitions and properties of I/O automata (e.g., composition and fairness) can be used directly in descriptions and proofs of shared memory algorithms.

3.1 Variables

We will model shared memory in terms of a collection of variables, so the first step is to define what is meant by a variable. We define a *variable* x to have a domain *dom(x)* of values and an initial value *init(x)* $\in$ *dom(x)*. Given a set X of variables, we model a state of X by an *assignment mapping* for X, denoted f_X, that maps each variable $x \in X$ to a value in *dom(x)*. We let F_X denote the set of all possible assignment mappings for X. We define *init(X)* to be the assignment mapping $f_X \in F_X$ such that $\forall x \in X, f_X(x) = init(x)$. If X and Y are sets of variables such that $Y \subseteq X$, we define $f_X|Y$ to be the assignment mapping $f_Y \in F_Y$ such that for all $y \in Y$, $f_Y(y) = f_X(y)$. If X and Y are disjoint sets of variables, and S_X, S_Y are sets of assignment mappings for X and Y, respectively, then we define $S_X \circ S_Y$ to be the set of assignment mappings S for $X \cup Y$ such that for all $s \in S, s|X \in S_X$ and $s|Y \in S_Y$. As shorthand, we may represent a singleton set of assignment mappings by its only element. For example, if f_X is an assignment mapping for X, we write $f_X \circ S_Y$ instead of $\{f_X\} \circ S_Y$. Analogously, for $f_X \in F_X$ and $f_Y \in F_Y$, we let $f_X \circ f_Y$ represent its only element when it is clear from context that a mapping (rather than a set of mappings) is called for. If $f \in F_X$, $x \in X$, and $v \in dom(x)$, we define $f_{[x=v]}$ to be the assignment mapping f' such that $f'|(X - \{x\}) = f|(X - \{x\})$ and $f'(x) = v$.

3.2 Shared Memory Actions

Since the only "sharing" that occurs in the I/O automaton model is the sharing of actions, we construct shared memory on top of the existing shared action mechanism. We begin by defining a special type of action called a "shared memory action" that will be used to model accesses to the shared variables. In some sense, this is the reverse of what is often done to incorporate message passing into a shared memory model. In UNITY [4], for example, shared queue variables are declared to model "channels" and atomic accesses to these shared queues model "sending" and "receiving" data across the channels. However, we emphasize that we do *not* implement shared memory on top of message passing. Rather, both are modelled directly in terms of the shared action mechanism.

We fix $\mathcal{L}$, a universal set of *access labels.* Let X be a set of variables. We define a *shared memory action for* X to be a triple of the form (f'_X, a, f_X), where $f'_X, f_X \in F_X$ and $a \in \mathcal{L}$.[3] We let *sm-acts(X)* denote the set of all possible shared memory actions for X. We say that π is a *shared memory action* iff it is a shared memory action for some X. We say σ is a shared memory step (for X) iff its contained action is a shared memory action (for X).

To construct signatures for shared memory automata, we need the following technical definition. Let Π be a set of actions and X a set of variables. We say that Π is *complete* for X iff $\forall \pi \in \Pi$, if $\pi = (f'_X, a, f_X)$ is a shared memory action for X, then $\forall \hat{f}'_X, \hat{f}_X \in F_X, (\hat{f}'_X, a, \hat{f}_X) \in \Pi$.

Let X and Y be sets of variables such that $Y \subseteq X$. If $\pi = (f'_X, a, f_X)$ is a shared memory action for X, we define its projection on Y, denoted $\pi|Y$, to be $(f'_X|Y, a, f_X|Y)$, a shared memory action for Y. If β is a sequence of actions, all of whose shared memory actions are shared memory actions for X, then we define $\beta|Y$ to be the sequence that results from replacing each shared memory action of β by its projection on Y. Projections on sets of shared memory actions, signatures

[3]These triples are action names, not to be confused with the steps of an automaton.

containing shared memory actions, and sets of sequences containing shared memory actions are defined analogously. If $\sigma = (s', \pi, s)$ is a step with π a shared memory action for X, then $\sigma|Y$ is defined to be $(s', \pi|Y, s)$.

3.3 Shared Memory Automata

Let X be a set of variables, and let A be an I/O automaton all of whose shared memory actions are external shared memory actions for X. Let *shared-in*(A) denote the set of shared memory actions that are inputs to A, and let *shared-out*(A) denote the shared memory actions that are outputs of A. We say that A is a *shared memory automaton for* X iff it satisfies the following conditions:

1. The sets of actions *shared-in*(A) and *shared-out*(A) are each complete for X.
2. For all steps $(s', (f'_X, a, f_X), s) \in steps(A)$,
   if $(f'_X, a, f_X) \in$ *shared-out*(A), then for all $\hat{f'_X} \in F_X$, there exists a state $\hat{s}$ and some $\hat{f_X} \in F_X$ such that $(s', (\hat{f'_X}, a, \hat{f_X}), \hat{s}) \in steps(A)$.
3. In the equivalence relation $part(A)$, any two output actions (f'_X, a, f_X) and $(\hat{f'_X}, a, \hat{f_X})$ are elements of the same equivalence class.

The first condition says that if A has a shared memory action with a given label a, then it has all possible shared memory actions with label a. For input actions, this means that A must be prepared to handle any value it may observe in the shared variables (since inputs are always enabled). For output actions, this condition is simply a technical restriction that makes composition of shared memory automata work out properly, as we will see later. The condition also makes describing the signatures of shared memory automata more convenient, since we need not list all the allowable values of the shared variables for each shared memory action label used.

The second condition says that for each shared memory output step, there exists a step from the same state for each possible assignment of the shared variables. In essence, this says that the preconditions of an output action may not depend on the values of the shared variables. This corresponds with the notion that one cannot observe the values of shared variables except by accessing them, and that one must be prepared to handle any value that might be observed.

The third condition says that the equivalence class membership of an output action may not depend upon the values of the external variables. This is a technical condition that prevents a nonsensical situation in which executions must be "fair" to the different values of the shared variables.

Since a shared memory automaton is an I/O automaton, all the standard I/O automaton definitions for executions, schedules, behaviors, composition, and fairness carry over to shared memory automata.

Theorem 2: The composition of a strongly compatible collection of shared memory automata for X is a shared memory automaton for X.

Proof: We know that the composition of a strongly compatible collection of I/O automata is an I/O automaton. Furthermore, since external actions of the components are external actions of the composition, we know that all of the shared memory actions are external actions in the composition. All of these are shared memory actions for X. It remains to be shown that the composition satisfies the three conditions imposed on shared memory automata for X. Condition 1 holds, since the union of complete sets of actions is clearly a complete set. For condition 2, we note that composition does not introduce any new output actions, nor does it remove any existing output actions. Furthermore, input-enabling and the definition of composition imply that for each output step (s'_i, π, s_i) of a component $\mathcal{A}_i$, for all states s' of the composition $\mathcal{A}$, if $s'|\mathcal{A}_i = s'_i$, then there exists a state s of $\mathcal{A}$ such that (s', π, s) is a step of $\mathcal{A}$. Thus, Condition 2 holds. Since the equivalence relation of the composition is the union of the individual equivalence relations of the components, any two actions in the same equivalence class in a component are in the same equivalence class in the composition. Since the set of shared memory output actions for each component is complete, strong compatibility assures us that no two shared memory output actions with the same label occur in different classes of the composition. This guarantees Condition 3. ∎

So far, we have given a general set of definitions for modelling collections of modules that access shared memory. Our accesses allow a module to atomically read the entire contents of memory, perform some local computation (possibly resulting in a state transition), and update the entire contents of shared memory. This general type of shared memory access is, of course, an expensive operation to implement. Therefore, we would like to define systems in which the shared memory accesses are more restricted. For example, in the most restricted case, we might only allow read or write accesses to single shared variables.

Let A be a shared memory automaton for X, let a be an access label of A, and let $x \in X$. We say that a is a

1. *read access to x* iff $\forall (s', (f', a, f), s) \in \mathit{steps}(A)$,
   (a) $f = f'$ and
   (b) $\forall \hat{f} \in F_X$ such that $\hat{f}(x) = f'(x)$, $(s', (\hat{f}, a, \hat{f}), s) \in \mathit{steps}(A)$.
2. *write access to x with value v* iff $\forall (s', (f', a, f), s) \in \mathit{steps}(A)$,
   (a) $f = f'_{[x=v]}$ and
   (b) $\forall \hat{f} \in F_X$, $(s', (\hat{f}, a, \hat{f}_{[x=v]}), s) \in \mathit{steps}(A)$.

In a read access to x, the shared memory is unmodified and the new state of A depends only upon the value observed in variable x. In a write access to x, the "before" and "after" states of shared memory differ only in the value of variable x, and the new state of A and the new value of x are independent of the "before" state of shared memory.

We now define a restricted class of shared memory automata called "single-variable read-write automata." In such automata, each access label for a shared memory output is constrained to be a read access or a write access to a single variable. Let A be a shared memory automaton for X, and let ψ be a partition of the access labels for actions in *shared-out*(A) such that there exist exactly two classes in ψ for each variable in $x \in X$, denoted $\psi_r(x)$ and $\psi_w(x)$. The partition ψ is called the *access partition* of A. We say that A is a *single-variable read-write automaton under ψ* iff $\forall x \in X$, $\psi_r(x)$ contains only read accesses to x and $\psi_w(x)$ contains only write accesses to x. We say that such an automaton *can read x* iff $\psi_r(x)$ is nonempty, and *can write x* iff $\psi_w(x)$ is nonempty. If Q is a collection of single-variable read-write automata, then a component of Q is said to *own* a variable x if it is the only component that can write x; in this case, x is said to be a *single-writer* variable. *Multi-writer*, *single-reader*, and *multi-reader* variables are defined in the obvious way.

Other classes of shared memory automata could be constructed in a similar manner. For example, one might define test-and-set or memory-to-memory-swap accesses and define automata in which the access labels are appropriately partitioned into additional classes. In fact, this style of definition can be used to define shared memory accesses for operations on arbitrary data types, such as enqueue and dequeue. Of course, any shared memory algorithm could be expressed and studied using the general shared memory automaton definition only, but being specific about the types of shared memory accesses allowed makes the assumptions about the underlying shared memory more explicit, and also may help simplify reasoning about the algorithm.

Notice that by exposing the values of the shared variables as part of the shared memory accesses, we not only carry forward the compositionality properties of I/O automaton behaviors but also provide a useful notion of a shared memory action as an input. We expect normal communication through shared variables to be modeled using output actions only, but the input actions allow a module to passively observe the accesses to shared memory made by other processes. We see two potential uses for this feature. First, one might use shared memory actions as inputs to construct external processes that are not part of the algorithm but monitor the use of shared memory (possibly as a means to check algorithms in a simulation system). Second, in a modular algorithm design, it may be appropriate to divide a task into several I/O automaton components such that only one component accesses the shared memory while the others are kept "informed" of these accesses by receiving them as inputs (e.g., to model a collection of processes "snooping" on a memory bus to update local caches).

3.4 Augmentation and Augmented-Composition

In building up I/O automaton systems, we may wish to compose collections of shared memory automata having different (either intersecting or disjoint) sets of shared variables. We would like the result of this composition to be a shared memory automaton for Z, where Z is the union of the sets of shared variables of the automata being composed. In order to accomplish this, we first "augment" each of the automata with additional shared variables so that its set of shared variables is Z. Then we compose as usual.[4]

We now define what is meant by augmenting an automaton. Let X and Z be sets of variables, with $X \subseteq Z$. Given a shared memory automaton A for X, we define *augment*(A, Z), read "the augmentation of A to Z," to be the automaton B as follows:

- $in(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}in(A)\} \cup$ $(in(A) - shared\text{-}in(A))$.
- $out(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}out(A)\} \cup$ $(out(A) - shared\text{-}out(A))$.
- $int(B) = int(A)$.
- $states(B) = states(A)$.
- $start(B) = start(A)$.
- $steps(B)$ = all steps $\sigma = (s', \pi, s)$ such that either
  1. $\sigma \in steps(A)$ and π is not a shared memory action, or
  2. $\sigma|X \in steps(A)$ and $\pi \in shared\text{-}in(B)$, or
  3. $\sigma|X \in steps(A)$, $\pi = (f'_Z, a, f_Z) \in shared\text{-}out(B)$, and $f'_Z|(Z - X) = f_Z|(Z - X)$.
- $part(B) = \{C \subseteq local(B) : C|X \in part(A)\}$ such that $part(B)$ forms a partition of the locally-controlled actions of B.

[4]When composing a shared memory automaton with an "ordinary" I/O automaton, no augmentation is necessary, since an ordinary I/O automaton is by definition an SMA for any set of variables X.

Essentially, we augment A by making the signature complete for Z, while leaving the set of states unchanged. For each step involving a shared memory action π for X, we substitute the set of all steps in which π is replaced by a shared memory action for Z (call it π') such that $\pi'|X = \pi$. For output actions steps, we make the further restriction that if $\pi' = (f'_Z, a, f_Z)$, then f'_Z and f_Z differ only in their assignments to the variables of X. This models the fact that outputs of B only change the values of shared variables in X. We do not make this restriction for input actions because they are always enabled. This also highlights the fact that the shared memory accesses of B are independent of all shared variables other than those in X. The partition of B is constructed from that of A to reflect the differences in their signatures.

Theorem 3: Let X and Z be sets of variables, with $X \subseteq Z$, and let A be a shared memory automaton for X. Then *augment*(A, Z) is a shared memory automaton for Z.

Proof: Immediate from the definitions of augmentation and shared memory automata. ∎

Our next result, Theorem 6, says that augmentation does not (in any significant way) affect the behavior of an automaton. This is proved using the following lemmas.

Lemma 4: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X and α_A is an execution of A, then there exists an execution α_B of $B =$ *augment*(A, Z) such that $\alpha_B|X = \alpha_A$.

Proof: Clearly, if α_A contains no actions, the claim holds. For the inductive hypothesis, let $\alpha_A = \alpha'_A \pi_A s$ be an execution of A, and let α'_B be the execution of B such that $\alpha'_B|X = \alpha'_A$. Clearly the state of A after α'_A is the same as the state of B after α'_B. Let this state be s'. It remains to be shown that some π_B is enabled from s' in B, resulting in state s, where $\pi_B|X = \pi_A$. If π_A is not a shared memory action, then the result is trivial, since the steps of A and B differ only with respect to shared memory actions. If π_A is a shared memory action (f'_X, a, f_X), then by the definition of augmentation, there must be a step $(s', \pi_B = (f'_Z, a, f_Z), s) \in$ *steps*(B) such that $\pi_B|X = \pi_A$. ∎

Lemma 5: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X and α_B is an execution of $B =$ *augment*(A, Z), then there exists an execution α_A of A such that $\alpha_A = \alpha_B|X$.

Proof: If α_B has no actions, the claim holds. For the inductive hypothesis, let $\alpha_B = \alpha'_B \pi_B s$ be an execution of B, and let α'_A be the execution of A such that $\alpha'_A|X = \alpha'_A$. Clearly the state of B after α'_B is the same as the state of A after α'_A. Let this state be s'. It remains to be shown that some π_A is enabled from s' in A, resulting in state s, where $\pi_A = \pi_B|X$. If π_B is not a shared memory action, then the result is trivial as before. If π_B is a shared memory action (f'_Z, a, f_Z), then by the definition of augmentation, the step $(s', (f'_Z|X, a, f_Z|X), s) \in$ *steps*(A). Therefore, the second claim holds. ∎

Theorem 6: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for X, then

1. *behs*$($*augment*$(A, Z))|X =$ *behs*(A), and
2. *fairbehs*$($*augment*$(A, Z))|X =$ *fairbehs*(A).

Proof: Part 1 is immediate from Lemmas 4 and 5.

For Part 2, let α_A be a fair execution of A, and let $\beta_A =$ *beh*(α_A). From Lemma 4, we know that there exists an execution α_B of $B =$ *augment*(A, Z) such that $\alpha_B|X = \alpha_A$. To show that

α_B is fair, we apply the definition of augmentation. From the construction of *steps*(B), a shared memory action $\pi \in$ *acts*(B) is enabled in state s of B only if $\pi|X$ is enabled in state s of A. The remaining actions $\pi \in$ *acts*(B) are enabled in in state s of B only if π is enabled in state s of A. Furthermore, any two actions π and π' are in the same equivalence class of B iff $\pi|X$ and $\pi'|X$ are in the same equivalence class of A. So, since α_A is fair, α_B is fair.

Now, to show the other direction, let α_B be a fair execution of B. By Lemma 5, there exists an execution α_A of A such that $\alpha_A = \alpha_B|X$. To show that α_A is fair, we argue similarly to above. ■

We can now define augmented–composition, making use of the augmentation definition and standard I/O automaton composition.

Augmented–Composition: Let $\{X_i\}_{i\in I}$ be a collection of (not necessarily disjoint) sets of variables, let $Z = \cup_{i\in I}X_i$, let each A_i be a shared memory automaton for X_i, and let the collection $\{augment(A_i)\}_{i\in I}$ be strongly compatible. We define the *augmented composition* $\prod^{+}_{i\in I} A_i$ to be the ordinary I/O automaton composition $\prod_{i\in I} augment(A_i, Z)$.

Theorem 7: Let $\{X_i\}_{i\in I}$ be a collection of (not necessarily disjoint) sets of variables, let $Z = \cup_{i\in I}X_i$, let each A_i be a shared memory automaton for X_i, and suppose that the collection of automata $\{augment(A_i, Z)\}_{i\in I}$ is strongly compatible. Then the augmented composition $\prod^{+}_{i\in I} A_i$ is a shared memory automaton for Z.

Proof: By Theorem 3, for each A_i, *augment*(A_i, Z) is a shared memory automaton for Z. Therefore, by Theorem 2, the result holds. ■

The following three compositionality results follow immediately from the corresponding results in [16], together with Theorems 6 and 7. The first result says that an execution of an augmented-composition induces executions of the component shared memory automata.

Corollary 8: Let $\{X_i\}_{i\in I}$ be a collection of sets of variables, where $Z = \cup_{i\in I}X_i$. Let $\{A_i\}_{i\in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i\in I}$ be strongly compatible, and let $A = \prod^{+}_{i\in I} A_i$. If $\alpha \in$ *execs*(A) then $(\alpha|augment(A_i, Z))|X_i \in$ *execs*(A_i) for every $i \in I$. Moreover, the same result holds if *execs*() is replaced by *fairexecs*(), *scheds*(), *fairscheds*(), *behs*(), or *fairbehs*().

The next result says that executions of component shared memory automata can often be pasted together to form an execution of the augmented-composition.

Corollary 9: Let $\{X_i\}_{i\in I}$ be a collection of sets of variables, where $Z = \cup_{i\in I}X_i$. Let $\{A_i\}_{i\in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i\in I}$ be strongly compatible, and let $A = \prod^{+}_{i\in I} A_i$. Suppose α_i is a (fair) execution of A_i for every $i \in I$, and let β be a sequence of actions in *acts*(A) such that $(\beta|augment(A_i, Z))|X_i =$ *sched*(α_i) for every $i \in I$. Then there is a (fair) execution α of A such that $\beta =$ *sched*(α) and $\alpha_i = (\alpha|augment(A_i, Z))|X_i$ for every $i \in I$. Moreover, the same result holds when *acts*() and *scheds*() are replaced by *ext*() and *beh*().

Finally, schedules and behaviors of component shared memory automata can also be pasted together to form schedules and behaviors of the augmented-composition.

Corollary 10: Let $\{X_i\}_{i\in I}$ be a collection of sets of variables, where $Z = \cup_{i\in I}X_i$. Let $\{A_i\}_{i\in I}$ be a collection of automata such that each A_i is a shared memory automaton for X_i. Let the collection of automata $\{augment(A_i, Z)\}_{i\in I}$ be strongly compatible, and let $A = \prod^{+}_{i\in I} A_i$. Let β

be a sequence of actions in $acts(A)$. If $(\beta | augment(A_i, Z)) | X_i \in scheds(A_i)$ for every $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds when $acts()$ and $scheds()$ are replaced by $ext()$ and $behs()$, respectively, and similarly when replaced by $acts()$ and $fairscheds()$ or by $ext()$ and $fairbehs()$.

3.5 The Closeout Operator

So far, we have introduced shared memory actions to model accesses to shared variables, and we have defined a special kind of I/O automaton containing shared memory actions in its signature. We have interpreted the first triple of each action as the "before state" of shared memory and the third component as the "after state." However, we have not yet placed any restrictions on the the relationship between the "after state" of one shared memory action and the "before state" of the next. A shared memory automaton is *not* guaranteed that the value it writes to a given shared variable will be the value observed by the next system component reading that variable. In other words, we permit the environment to freely modify the values in shared memory. We would like to construct systems in which the set of components that may modify a particular shared variable is fixed, closed to the environment. We therefore define a "closeout" operator, which takes a shared memory automaton A and produces a new automaton B such that some or all of the shared variables of A become part of the local state of B. In this way, the "absorbed" variables can be touched only the by the actions of B. Since A may be the result of composing several shared memory automata, the closeout operator achieves the desired result of restricting shared variable accesses to a particular collection of system modules.

We now define the closeout operator $\mathcal{C}$. Since the state of an automaton may be thought of as a mapping from a set of variables to a set of values, we will feel free to operate on states as if they were assignment mappings. Let X and Y be disjoint sets of variables, let $Z = X \cup Y$, and let A be a shared memory automaton for Z. We define $B = \mathcal{C}(A, X)$ as follows:

- $sig(B) = sig(A)|Y$
- $states(B) = states(A) \circ F_X$,
- $start(B) = start(A) \circ init(X)$,
- $steps(B)$ contains exactly the following set of steps: for each step (s', π, s) in $steps(A)$,
  1. if $\pi = (f'_Z, a, f_Z)$ is a shared memory action, then
     $(s' \circ (f'_Z|X), (f'_Z|Y, a, f_Z|Y), s \circ (f_Z|X)) \in steps(B)$,
  2. if π is not a shared memory action, then
     $\{(s' \circ f_X,\ \pi,\ s \circ f_X) : f_X \in F_X\} \subseteq steps(B)$, and
- $part(B) = part(A)$, where each class is projected on Y.

Essentially, the variables in X are absorbed into the internal state of the "closed out" automaton. If $x \in X$, we use the familiar record notation $s.x$ to refer to the value of x in a particular state s of B. That is, if $s_B = s_A \circ f_X$, where s_A is a state of A, then $s_B.x = f_X(x)$.

Given the definition of the closeout operator, we get the following natural result.

Theorem 11: Let A be a shared memory automaton for Z and let X and Y be disjoint sets of variables such that $Z = X \cup Y$. Then $B = \mathcal{C}(A, X)$ is a shared memory automaton for Y.

Proof: To show that B is an I/O automaton, we must demonstrate that for all states s' and input actions π of B, there exists a state s' of B such that $(s', \pi, s) \in steps(B)$. Since this property is true of A, and since *shared-in*(A) is complete, this property is also true of B by the construction of *steps*(B). (When we construct the steps of B, completeness of *shared-in*(A) guarantees that we include all possible values for X in the "before states" of the steps for each input action.)

We now show that I/O automaton B is a shared memory automaton for Y. Clearly, all the shared memory actions of B are external shared memory actions for Y. We now show that each of the three conditions in the definition of a shared memory automaton hold for B. For the first condition, since *shared-in*(A) is complete for Z, *shared-in*(B) = *shared-in*$(A)|Y$ must be complete for Y. Similarly, for *shared-out*(B). The second condition requires that for every step $(s', (f'_Y, a, f_Y), s)$ in *steps*(B), if $(f'_Y, a, f_Y) \in$ *shared-out*(B), then for all $\hat{f'_Y} \in F_Y$, there exists a state $\hat{s}$ and some $\hat{f}_Y \in F_Y$ such that $(s', (\hat{f'_Y}, a, \hat{f}_Y), \hat{s})$ is in *steps*(B). Since this condition is true for A, we know that for each shared memory output action label a, there exists a step $(s',\ (f'_X \circ f'_Y, a, f_X \circ f_Y), s)$ for every possible assignment mapping $f'_X \circ f'_Y$ for Z. Therefore, when we project on Y in constructing *steps*(B), we have a step $(s',\ (f'_Y, a, f_Y), s)$ for each possible assignment mapping f'_Y for Y. The third condition, regarding membership of equivalence classes, is obviously true of B. ∎

3.6 Closeout for behaviors

We now give a closeout definition for behaviors that is analogous to the one for automata.

Let X and Z be sets of variables with $X \subseteq Z$. If β is a sequence of actions of a shared memory automaton A for Z, then we say that β is *consistent for* X iff the following conditions hold:

1. if (f'_Z, a, f_Z) is the first shared memory action in β, then $f'_Z|X = init(X)$, and
2. if (f'''_Z, a_1, f''_Z) and (f'_Z, a_2, f_Z) are shared memory actions in β such that no shared memory action occurs between them, then $f''_Z|X = f'_Z|X$.

If Σ is a set of sequences of actions of a shared memory automaton for Z, then we define $\mathcal{C}(\Sigma, X)$ to be the set $\Sigma_X|(Z - X)$, where Σ_X is the subset of Σ containing exactly those sequences that are consistent for X.

Lemma 12: Let X and Z be sets of variables such that $X \subseteq Z$. Let A be a shared memory automaton for Z, and let α_B be an execution of $B = \mathcal{C}(A, X)$ with behavior β_B. Then there exists an execution α_A of A, with behavior β_A consistent for X, such that $\beta_A|(Z - X) = \beta_B$.

Proof: Let $Y = Z - X$. We construct the sequence α_A from α_B as follows. For each step $(s' \circ f'_X, \pi, s \circ f_X)$ in α_B, if $\pi = (f'_Y, a, f_Y)$ is a shared memory action of B, then we let the corresponding step in α_A be $(s', (f'_Y \circ f'_X, a, f_Y \circ f_X), s)$; and if π is not a shared memory action, we let the corresponding step in α_A be (s', π, s).

Let $\beta_B = beh(\alpha_A)$. Clearly, $\beta_B|Y = \beta_A$. It remains to be shown that α_A is an execution of A and that β_A is consistent for X. We show that α_A is an execution of A by showing that each step of α_A is in *steps*(A). Let $\sigma = (s' \circ f'_X, \pi, s \circ f_X)$ be a step of B. If $\pi = (f'_Y, a, f_Y)$ is a shared memory action of B, then by the construction of *steps*(B) in the definition of closeout, $(s', (f'_Y \circ f'_X, a, f_Y \circ f_X), s)$ must be a step of A. Similarly, if π is not a shared memory action, then (s', π, s) must be a step of A. Therefore, the construction produces an execution of A.

Finally, we show that β_A is consistent for X. Since every initial state of $\mathcal{C}(A, X)$ is in $states(A) \circ init(X)$, it must be that the first shared memory action (f'_Z, a, f_Z) of β_B has $f'|X = init(X)$, so the first consistency condition is satisfied. We know that the second consistency condition must be satisfied, since any two successive steps (s''', π_1, s'') and (s', π_2, s) of any execution must have $s'' = s'$, the assignments to the variables of X are part of the state of $\mathcal{C}(A, X)$, and the only actions that may change the values for X in the state of $\mathcal{C}(A, X)$ correspond to shared memory actions for for Z. ∎

Lemma 13: Let X and Z be sets of variables such that $X \subseteq Z$. Let A be a shared memory automaton for Z and let α_A be an execution of A with behavior β_A. If β_A is consistent for X, then there exists an execution α_B of $B = \mathcal{C}(A, X)$ such that $\beta_A|(Z - X)$ is the behavior of α_B.

Proof: Let $Y = Z - X$. Let α_B be the execution constructed from α_A as follows. For each shared memory action π in α_A, let the corresponding action in α_B be $\pi|Y$. Leave the remaining actions as in α_A. For each state s in α_A, let the corresponding state in α_B be $s \circ (f_Z|X)$, where f_Z is the third component of the preceding shared memory action in α_A (or $f_Z = \mathit{init}(Z)$ if there is no preceding shared memory action).

Clearly $\beta_A|Y = \mathit{beh}(\alpha_B)$. We claim that α_B is an execution of B. To prove this claim, we proceed by induction on the length of α_B, showing that each action is enabled from the state in which it occurs. Clearly, if α_B contains no actions, then the claim holds. Let (s'_A, π, s_A) be a step of α_A, and let α'_B be the portion of α_B up to (but not including) the action $\pi|Y$ for the corresponding step in α_B. We wish to show that if α'_B ends in state s'_B, then the step $(s'_B, \pi|Y, s_B) \in \mathit{steps}(B)$, where s_B is the next state of α_B. By the construction, we know that $s'_B = s'_A \circ (f'_Z|X)$, where f'_Z is the third component of the preceding shared memory action in α_A (or $f'_Z = \mathit{init}(Z)$ if there is no preceding shared memory action), and similarly for s_B. There are two cases for π:

1. If π is not a shared memory action, then clearly it is enabled from s'_B, since (by the construction) s'_A and s'_B are identical except that s'_A does not assign values to the variables in X. Furthermore, since π is not a shared memory action, $s_B|X = s'_B|X$, so the step exists by the definition of the closeout operator.

2. If $\pi = (f'_Z, a, f_Z)$ is a shared memory action, then consistency of β_A requires that f'_Z be the third component of the preceding shared memory action in α_A (or $\mathit{init}(Z)$ if there is no such preceding action). By the definition of closeout, we know $\mathit{steps}(B)$ contains the step $(s'_A \circ (f'_Z|X),\ (f'_Z|Y, a, f_Z|Y),\ s_A \circ (f_Z|X))$. And by the construction, $s'_A \circ (f'_Z|X) = s'_B$ and $s_A \circ (f_Z|X) = s_B$. Therefore, the desired step exists.

In both cases, $\pi|Y$ is enabled and leads to state s_B. ∎

Theorem 14: Let X and Z be sets of variables such that $X \subseteq Z$. If A is a shared memory automaton for Z, then

1. $\mathit{behs}(\mathcal{C}(A, X)) = \mathcal{C}(\mathit{behs}(A), X)$, and

2. $\mathit{fairbehs}(\mathcal{C}(A, X)) = \mathcal{C}(\mathit{fairbehs}(A), X)$.

Proof: Part 1: Let $Y = Z - X$. By Lemma 12, we know that if $\beta|Y$ is a behavior of $\mathcal{C}(A, X)$, then β is a behavior of A that is consistent for X. Therefore $\beta|Y \in \mathcal{C}(\mathit{behs}(A), X)$, by definition. If $\beta|Y \in \mathcal{C}(\mathit{behs}(A), X)$, then by definition of closeout on behaviors, β is consistent for X. Therefore, Lemma 13 tells us that $\beta|Y \in \mathit{behs}(\mathcal{C}(A, X))$.

Part 2: First, we show that $\mathit{fairbehs}(\mathcal{C}(A, X))$ contains $\mathcal{C}(\mathit{fairbehs}(A), X)$. Let β_B be a fair behavior of $B = \mathcal{C}(A, X)$, and let α_B be an execution of B with $\mathit{beh}(\alpha_B) = \beta_B$. Construct execution α_A of A from α_B as in the proof of Lemma 12 such that $\mathit{beh}(\alpha_A)|(Z - X) = \beta_B$. Since A is a shared memory automaton, we know that $\mathit{shared\text{-}out}(A)$ is complete and that for any given access label $a \in \mathcal{L}$, all shared memory actions with label a belong to the same class. Furthermore, by the definition of closeout, π_A and π'_A belong to the same equivalence class in A iff $\pi_A|X$ and $\pi'_A|X$ belong to the same equivalence class in B. Therefore, given that α_B is fair, we can show that α_A is fair by arguing that an action π_A is enabled in state s_A of α_A iff $\pi_A|X$ is enabled in the corresponding state s_B of α_B. This is easily seen from the construction of $\mathit{steps}(B)$, since $s_A = s_B|(Z - X)$.

Now, we show that $\mathcal{C}(\mathit{fairbehs}(A), X)$ contains the set $\mathit{fairbehs}(\mathcal{C}(A, X))$. Let β_A be a fair behavior of A that is consistent for X, and let α_A be an execution of A with $\mathit{beh}(\alpha_A) = \beta_A$. Construct execution α_B of $\mathcal{C}(A, X)$ from α_A as in the proof of Lemma 13 such that $\beta_A|(Z - X) = \mathit{beh}(\alpha_B)$. The remainder of the proof is argued as above. ∎

4 Proofs for Shared Object Systems

It is convenient to use shared atomic objects as system components when building large concurrent systems. Each operation on an atomic object appears to execute indivisibly, thereby allowing the programmer to consider only interleavings of the operations, rather than their true concurrency. For performance reasons, however, it is often useful to allow concurrency between operations on a single object, so the condition of atomicity is only on an object's behavior, not on its implementation.

There are two general approaches to modelling shared objects. Which is more natural depends upon whether the intent is to model a system that *uses* atomic objects, or one that *implements* them. In a system that uses atomic objects, it is convenient to represent each operation as a single shared action between the object and the invoking process; we call these *atomic access systems.* In a system that implements atomic objects, each operation can be modeled by an invocation event and response event, denoting, respectively, the beginning and end of operation execution; we call these *invocation-response systems.*

In an invocation-response system, it is possible to consider operation executions that overlap in time. Herlihy and Wing [8] use this model to define a correctness condition called *linearizability* that extends Lamport's notion of atomicity for reads and writes [11] to arbitrary data types. Linearizability requires that in any (concurrent) execution, each operation "appears" to take effect instantaneously, sometime between the invocation and response events of the operation. Linearizability is also similar to Lamport's *sequential consistency* [10], but requires that if two operations on a given object do not overlap in time, then the order in which they "appear" to occur is consistent with the order in which they actually occur [5]. Furthermore, unlike sequential consistency, linearizability has a locality property: if each object is linearizable, then the entire system is linearizable.

In this section, we take the linearizability notion one step further, and show that a linearizable invocation-response system is equivalent to an atomic access system. In particular, we show that if the objects in the invocation-response system are each *linearizable*, then every behavior of the entire invocation-response system is a behavior of the atomic access system. Thus, in reasoning about complex systems, it is possible to consider overlapping operation executions at one level of abstraction, and shared atomic actions for the same operations when reasoning at higher levels of abstraction. This is an important benefit of using a model that unifies invocation-response and atomic access in a single formal framework. In addition, we extend the work of Herlihy and Wing by treating not only safety properties of invocation-response systems, but liveness properties as well.

These results are intended to be used for reasoning about multiprocessor programs, in which the shared memory is assumed to be atomic and the program is modularized by layers of linearizable concurrent objects. They may also be useful in reasoning systems having a mixture of message passing and shared memory operations. For example, a network of shared memory multiprocessors or a message passing multiprocessor with multi-threaded nodes would involve message passing between nodes and shared memory operations within a node.

We begin, in Section 4.1 by describing the basic architecture of an invocation-response system, including the interfaces and specification mechanisms for the objects and processes. Then, in

[5]A comparison of sequential consistency and linearizability is given by Attiya and Welch [2].

Section 4.2, we describe three systems: a concurrent system, a sequential system, and an atomic system. All three systems are described in the I/O automaton model. We show that if the objects of the concurrent system are linearizable and if the processes obey certain well-formedness restrictions, then each of these systems "simulates" the next. This gives us a unified theory for describing systems in terms of invocations and responses, but reasoning about them in terms of atomic accesses.

4.1 Invocation-Response Systems

We are interested in studying systems in which processes invoke operations on objects and then wait for the objects to respond. In this section, we define a general architecture for *invocation-response* systems (or *IR* systems). Later, this architecture will be used to define two systems: System C, a concurrent system containing linearizable objects, and System B, a sequential system used as a stepping stone in our proof. A different structure will be used to define System A, an atomic shared object system that will form the basis of our correctness condition.

An IR system consists of a set of processes and a set of objects, where each process and each object is modelled as an I/O automaton. Processes may request operations on objects by issuing "invoke" actions. These actions are inputs to the objects, which issue "respond" output actions after performing the requested operation. The interface at the boundary between a process and its environment is analogous to the interface at the boundary between an object and a process. To request that the system perform a particular function, the environment may "invoke" operations on a process, which later replies to the environment with a "respond" action. Here, we consider systems of only three layers: the objects, the processes, and the environment. However, by modelling the interaction between a process and the environment in the same way as the interaction between an object and a process, we set the stage for constructing complicated objects hierarchically. That is, one might compose a collection of objects and processes, and treat the composition as a single object. To describe the set of operations that may be invoked on an object or process, we define an "interface type." An *interface type* T consists of:

- $ops(T)$, a set of operation names, and
- for each operation $\rho \in ops(T)$,
  - $args(\rho)$, the domain of arguments to the operation, and
  - $rets(\rho)$, the domain of return values of the operation.

The operation names identify the operations that may be invoked on the corresponding object or process. For each operation name, the domain of argument values specifies the allowable operation arguments that may be supplied by the user of the object. Similarly, the return value domain for an operation specifies the possible values that may be returned by the object as a result of that operation. We will see shortly how the interface type of an object or process is used to derive the signature of the corresponding automaton.

In IR systems, there are three kinds of components: the shared objects, the processes that invoke operations on those objects, and the environment that directs the activities of the processes. For the remainder of the paper, we fix three sets of indices, $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{K}$. We use the elements of $\mathcal{I}$ to name the objects in a system, and we use the elements of $\mathcal{J}$ to name the processes that invoke operations on the objects. An IR system is modelled as the composition of an object automaton o_i for each $i \in \mathcal{I}$, and a process automaton p_j for each $j \in \mathcal{J}$. The indices in $\mathcal{K}$ identify the processes (or users) that constitute the environment of a system.

An example IR system is shown in Figure 1. Objects (in $\mathcal{I}$) are shown as squares, processes (in $\mathcal{J}$) are circles, and components of the environment (in $\mathcal{K}$) are triangles. We do not model the

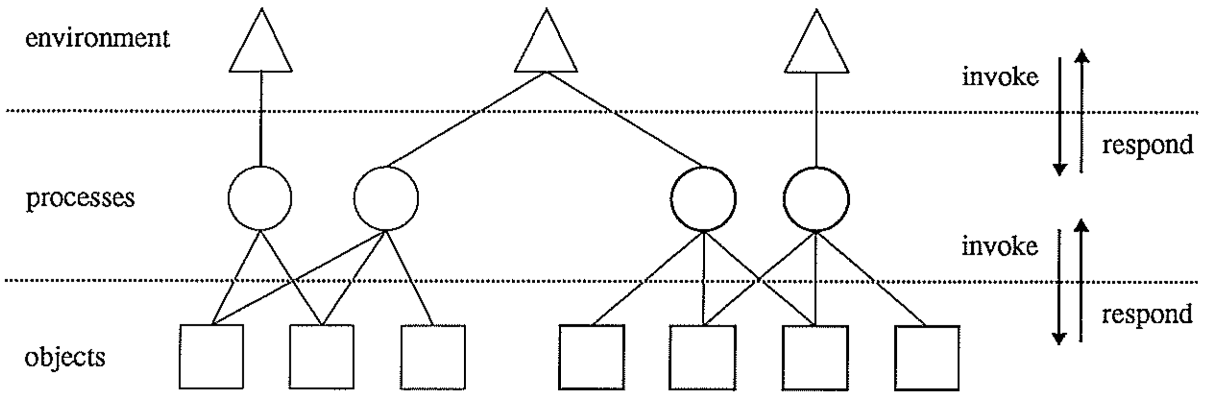

Figure 1: An IR System.

environment explicitly, but simply use the elements of $\mathcal{K}$ to refer to its components. One may think of these components either as I/O automata or as users interacting with the system. We require that the environment, as a whole, obey certain well-formedness restrictions on its interactions with each process. Informally, we require that the environment wait for a process to respond to a request before making a new request of that process.[6] If several components in the environment may make requests of the same process, then those components must cooperate (possibly by participating in a mutual exclusion protocol) in order to ensure that well-formedness is preserved at that process. We now define the objects and processes of IR systems.

4.1.1 Objects

Each object automaton $o_i, i \in \mathcal{I}$, has an associated interface type, denoted *type*(i), and the signature of o_i is determined from this interface type. For each $i \in \mathcal{I}$, we define $sig(o_i)$ as follows:

Input Actions:	$\text{invoke}_{i,j}(\rho, a)$, $j \in \mathcal{J}$, $\rho \in ops(type(i))$ and $a \in args(\rho)$
Output Actions:	$\text{respond}_{i,j}(\rho, a, r)$, $j \in \mathcal{J}$, $\rho \in ops(type(i))$, $a \in args(\rho)$, and $r \in rets(\rho)$

The subscripts on each action identify the object automaton o_i at which the operation occurs and the process automaton p_j responsible for the request. The following definition is useful for describing executions of o_i. Let α be an execution of o_i. We say that α is *input well-formed* iff $\forall j \in \mathcal{J}$, no two $\text{invoke}_{i,j}$ actions occur in α without a $\text{respond}_{i,j}$ action between them. Input well-formedness does not prohibit concurrency at an object, since multiple invocations from *different* elements of $\mathcal{J}$ may occur without intervening responses.

4.1.2 Processes

Each process in an IR system is modelled as an I/O automaton p_j, $j \in \mathcal{J}$, that has an associated interface type, denoted *type*(j). The interface type describes the set of operations that may be invoked on a process. In addition, a process may itself invoke operations on objects. Therefore, its signature not only contains actions corresponding to operations in its interface type, but also invoke and respond actions corresponding to the interface types of the objects that it may access. For each $j \in \mathcal{J}$, we let *obj(j)* $\subseteq \mathcal{I}$ denote that set of objects that p_j may access, and define the signature of p_j as follows:

[6] A similar idea appears in [14] on page 79.

Input Actions:	invoke$_{j,k}(\rho, a)$, where $k \in \mathcal{K}, \rho \in ops(type(j))$ and $a \in args(\rho)$
	respond$_{i,j}(\rho, a, r)$, where $i \in obj(j), \rho \in ops(type(i))$, $a \in args(\rho)$, and $r \in rets(\rho)$
Output Actions:	respond$_{j,k}(\rho, a, r)$, where $k \in \mathcal{K}, \rho \in ops(type(j))$, $a \in args(\rho)$, and $r \in rets(\rho)$
	invoke$_{i,j}(\rho, a)$, where $i \in obj(j), \rho \in ops(type(i))$ and $a \in args(\rho)$

In reasoning about the schedules of a process in an IR system, it will be helpful to distinguish those actions that are shared with the objects from those shared with the environment. Let β be a sequence of actions of p_j. We define $\beta|\mathcal{I}$ to be the subsequence of β containing exactly the invoke$_{i,j}$ and respond$_{i,j}$ actions, for all $i \in \mathcal{I}$ (the objects). Similarly, we define $\beta|\mathcal{K}$ to be the subsequence of β containing exactly the invoke$_{j,k}$ and respond$_{j,k}$ actions, for all $k \in \mathcal{K}$ (the environment).

As mentioned earlier, we constrain the interaction between each process and the environment so that the process receives no inputs from the environment while the process has an outstanding request. That is, we want the invocations and responses at the environment boundary of each process to alternate, where each response is appropriate for the preceding invocation. For this purpose, we use the following definition. If γ is a sequence of actions, $j \in \mathcal{J}$, and $\beta = \gamma|p_j$, we say that γ is *externally well-formed for j* iff $\beta|\mathcal{K}$ is an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall k \in \mathcal{K}, \forall \rho \in ops(P_j), \forall a \in args(\rho), \forall r \in rets(\rho)$, each respond$_{j,k}(\rho, a, r)$ action is immediately preceded by an invoke$_{j,k}(\rho, a)$ action. We say that γ is *externally well-formed* iff it is externally well-formed for all $j \in \mathcal{J}$. An execution is *externally well-formed* iff its schedule is well-formed.

In externally well-formed sequences, we think of a process as being "active" in the interval between receiving an invocation and generating a response. More formally, let β be an externally-well formed sequence of p_j, $j \in \mathcal{J}$. If β' is a prefix of β, we say that p_j *is active after* β' iff $\beta'|\mathcal{K}$ ends with an invoke action. It is important to notice that in externally well-formed executions, a process receives no inputs from the environment while it is active. However, multiple processes in $\mathcal{J}$ may be active at once, providing the environment with concurrent access to the underlying objects. In the example of Figure 1, each process is accessed by at most one component of the environment, so as long as each component of the environment preserves external well-formedness, so will the environment as a whole. A similar effect would be achieved by an implementation that dynamically created a new process for each invocation from the environment.

The processes in an IR system model the programs that access the objects on behalf of the environment, and their implementation depends upon the semantics needed for the application. Therefore, in stating the general definition of an IR system, we do not explicitly define the process automata. However, we do require that each p_j, $j \in \mathcal{J}$ preserves the following well-formedness condition. Let β be a sequence of actions, and let $\beta_j = \beta|p_j$, $j \in \mathcal{J}$. We say that β is *well-formed* for j iff the following conditions hold:

- β is externally well-formed for j.
- Every action in $\beta_j|\mathcal{I}$ occurs from a prefix of β_j after which p_j is active.
- The sequence $\beta_j|\mathcal{I}$ is an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall \rho \in ops(type(i)), \forall a \in args(\rho), \forall r \in rets(\rho)$, each respond$_{i,j}(\rho, a, r)$ action is immediately preceded in β_j by an invoke$_{i,j}(\rho, a)$ action.

So, in order to preserve well-formedness, p_j must respond at most once to each request from the environment, and it may invoke operations on objects only in the interval between a request from the environment and its response to that request. Furthermore, if p_j invokes an operation on an object, it may not produce any output actions until the corresponding response from that object occurs. Note that the third condition implies that p_j preserves input well-formedness in β for all $i \in \mathcal{I}$.

4.2 Simulating Atomic Access Systems with IR Systems

At the beginning of Section 4, we stated that an important problem in programming multiprocessor systems is to build IR systems containing concurrently accessed shared objects in such a way that the environment cannot distinguish them from atomically accessed shared memory systems. In this section, we take advantage of ability to study both shared memory and message-passing systems in the I/O automaton model in order to show a formal correspondence between systems containing concurrently accessed linearizable objects and systems having atomically accessed shared memory. We present three systems. The first is an IR system (System C) that models the system containing concurrently accessed linearizable objects. The second system, derived from the first, is an IR system (System B) in which at most one operation is in progress at each object at any time. Finally, we present an atomic access system (System A) that corresponds to system B, but implements the objects in atomically accessed shared memory. We show that the fair behaviors of System C are contained in those of System B, and that the fair behaviors of System B are contained in those of System A. This serves to formalize the notion that systems containing linearizable objects "simulate" those in which the objects are implemented in atomic memory. We begin with an overview of the three systems.

System C is the concurrent invocation-response system that we wish to prove simulates an atomic object system. It is an IR system, so each process of System C has an interface type, the appropriate signature for that type, and is required to preserve well-formedness. In addition, the processes must satisfy a property that implies that objects eventually respond to all operation requests. In order to define the objects of System C, we present a natural definition for a "sequential specification" of an object and define formally what it means for an object to be a "linearizable implementation" of such a specification. Each object of System C is described by a sequential specification and is constrained to be a linearizable implementation of that specification. Aside from the above restrictions, the processes and objects of System C are completely general. We do not use any information about the particular sequential specifications or implementations of the objects in order to prove our results. In this way, our results hold for any IR system with linearizable objects.

As we have said, our notion of correctness is that every fair behavior of System C should be a fair behavior of a system in which the objects are accessed atomically (as opposed to separate invocations and responses). In other words, System C should "simulate" a system in which the each operation on an object is implemented as a single atomic access to a shared memory. Rather than showing this simulation directly, we construct an intermediate system in which the processes are the same as in System C, but in which the objects are constructed explicitly from their sequential specifications. This intermediate system, called System B, is used as a stepping stone in the proof. We show that for every fair execution of System C, there is a fair execution of System B having the same external behavior, and in which each invocation of an operation on an object is immediately followed by the corresponding response.

Finally, we construct System A, the atomic system that forms the basis of our correctness condition. System A consists of a set of processes that perform atomic accesses on a shared memory. The system is constructed from the processes of System C and the sequential specifications of the objects. We show that for every fair execution of System B in which invocations are immediately followed by their corresponding responses, there is a fair execution of system A with the same behavior. The two simulation arguments (that System C simulates a certain class of executions of System B, and that those executions correspond to executions of System A) are combined to complete the proof.

4.2.1 System C

In this section, we define System C, the IR system that we wish to prove correct. System C is the composition of a collection of linearizable objects and processes that we wish to show behaves correctly. The automata in System C are not given to us explicitly, but are guaranteed to satisfy certain properties. From the definition of an IR system, we know that in System C each process automaton p_j, $j \in \mathcal{J}$, preserves well-formedness. Furthermore, we will assume that each object in System C is a "linearizable implementation" of a "sequential specification". In addition, we will make an assumption about liveness in System C. We begin by defining a sequential specification, and say what it means to be a linearizable implementation of one.

For each $i \in \mathcal{I}$, we fix a *sequential specification* S_i consisting of the following information:

- $states(i)$, a set of states containing a set of initial states $init(i)$.
- two predicates for each operation name $\rho \in ops(type(i))$:
  - predicate P_ρ on elements from $args(\rho) \times states(i)$, and
  - predicate Q_ρ on elements from $args(\rho) \times states(i) \times states(i) \times rets(\rho)$.

This means that if $a \in args(\rho)$ is the argument to operation ρ, $x' \in states(i)$ is the "current state" of object O, and the predicate P_ρ holds on a and x', then there exists an $x \in states(i)$ and an $r \in rets(\rho)$ such that $Q_\rho(a, x', x, r)$ is true. One such x becomes the current state of O following the operation, and r is returned by the operation. For arguments a and states x' for which $P_\rho(a, x')$ does not hold, the new state and return value are unspecified. In Larch specifications[3], this information is conveniently represented in the following way:

ρ = proc(a:$args(\rho)$) returns (r:$rets(\rho)$)
    requires: $P_\rho(a, x')$
    ensures: $Q_\rho(a, x', x, r)$

Having defined a sequential specification, we now wish to define what it means for the object automata of System C to be linearizable implementations of their sequential specifications.

Borrowing a technique from [12], we construct a particular automaton, called a "sequential object" that captures the meaning of a sequential specification. The sequential object construction will be used not only to define a linearizable implementation of a sequential specification, but also to define System B.

We capture the meaning of each sequential specification S_i, $i \in \mathcal{I}$, with a *sequential object automaton* a_i. The sequential object automaton a_i has signature $sig(o_i)$ and the following state components: $current \in states(i)$, $user \in \mathcal{J} \cup \perp$, $op \in ops(type(i)) \cup \perp$, and $arg \in \bigcup_{\rho \in ops(type(i))} args(\rho) \cup \perp$. The component *current* holds the "current state" of the object, and is initially in $init(i)$. The component *user*, initially $\perp$, is the index of the process currently using the object. Components *op* and *arg* hold the name and argument of the operation in progress; initially, these are both $\perp$. The transition relation for the sequential object automaton is given in Figure 2. "Pre" and "Post" denote precondition and postcondition, respectively. An action is enabled in exactly those states s' for which the precondition is satisfied. If an action has no precondition, it is enabled in all states. When an action occurs, p_i's new state s must satisfy the statements in the postcondition. States

- invoke$_{i,j}(\rho, a)$
  Post: $s.user = j$
  $s.op = \rho$
  $s.arg = a$

- respond$_{i,j}(\rho, a, r)$
  Pre: $s'.user = j$
  $s'.op = \rho$
  $s'.arg = a$
  $P_\rho(a, s'.current)$
  Post: $Q_\rho(a, s'.current, s.current, r)$
  $s.user = \perp$
  $s.op = \perp$
  $s.arg = \perp$

Figure 2: Transition relation for sequential object automaton a_i.

s and s' agree on components not explicitly constrained by the postcondition. The partition of a_i consists of a single class containing all the output actions of a_i.

When an invocation occurs at a_i, the automaton simply stores the id of the process making the request, the name of the operation, and the values of the arguments to the operation. Whenever an operation has been requested but the response has not yet occurred, a_i may respond to the request, supplying a return value consistent with the sequential specification and resetting the *user*, *op* and *arg* components to their initial values.

Next, we would like to define what it means for an automaton to be a linearizable implementation of sequential specification S_i. But first we need to define a particular class of executions of a_i. We say that α is a *sequential execution* of a_i iff $sched(\alpha)$ is an alternating sequence of invoke and respond actions.

We can now formally define linearizability. We say that o_i is a *linearizable implementation of* S_i iff for all input well-formed fair executions α of o_i, there exists a sequential fair execution α' of a_i such that for all $j, j' \in \mathcal{J}$,

1. $\alpha'|p_j = \alpha|p_j$, and
2. if a respond$_{i,j'}$ event π' precedes an invoke$_{i,j}$ event π in α, then π' precedes π in α'.

Informally, the first condition says that each individual process cannot distinguish α from α'. The second condition says that if the invocation-response intervals of two operations at an object do not overlap in α, then they must occur in the same relative order in α' as they do in α.

For every $i \in \mathcal{I}$, we require that o_i in System C is a *linearizable implementation of* S_i. Note that the linearizable implementation requirement is a local property of each object, and not a property of the system as a whole. We will consider global linearizability properties after defining System B.

Our final assumption about System C concerns liveness. We require that all externally well-formed executions γ of System C are *response-live*, meaning that for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, for all $\rho \in ops(type(i))$, for all $a \in args(\rho)$, if $\pi = invoke_{i,j}(\rho, a)$ occurs in γ, then there exists a state s after π such that some $respond_{i,j}$ action is enabled from each state after s until such an action occurs. With this assumption, we get the following liveness result:

Lemma 15: Let γ' be an externally well-formed fair execution of System C. Then for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, if an $invoke_{i,j}$ action occurs in γ then a $respond_{i,j}$ action occurs later in γ.

Proof: Immediate from the definitions of response-live and fairness. ∎

Notice that we could have imposed a condition stronger than response-liveness by prohibiting partial operations in sequential specifications entirely. (Prohibiting partial operations would ensure, by the definition of a_i and linearizability, that each object eventually responds to each request.) However, in order to allow modelling a class of systems in which the processes cooperate to ensure that operations are invoked only when appropriate, we choose to take a more general approach, in which the system must guarantee that a response eventually occurs for each request. For example, an object might be responsible for granting permission to use a shared resource (a lock) so that no two processes have permission simultaneously. Such an object could have two operations, one for requesting the lock and another for releasing it. If one process requests the lock while a second process process is holding the lock, then the object cannot respond to the request until the second process releases the lock. Thus, the operation is partial, but as long as processes are guaranteed to eventually release the lock, then all requests can be satisfied.

A special case of the response-live property is one that says that for all $i \in \mathcal{I}$, for all $j \in \mathcal{J}$, for all $\rho \in ops(type(i))$, for all $a \in args(\rho)$, if $\pi = invoke_{i,j}(\rho, a)$ occurs in γ, then some $respond_{i,j}$ action is enabled from each state after π until such an action occurs. In other words, if an object has partial operations, then (1) they are invoked only in states for which they are defined, and (2) if an operation is pending at a process, then no state change occurs to prevent a response to that operation. Although this property is stronger than the response-liveness property, it is a *safety* property and may be easier to prove when it is applicable.

4.2.2 System B

Rather than directly showing that System C simulates an atomic access system, it will be convenient to define an intermediate system, System B. We define System B to be identical to System C except that for all $i \in \mathcal{I}$, each object automaton o_i is replaced by a_i, the sequential object automaton corresponding to the sequential specification S_i.

Let β be an execution of System B. We say that β is a *sequential execution* of System B iff for all $i \in \mathcal{I}$, $\beta|a_i$ is a sequential execution of a_i and no actions occur in β within each invoke/respond interval of $\beta|a_i$. So, $\beta|\mathcal{I}$ consists of an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \forall \rho \in ops(type(i)), \forall a \in args(\rho), \forall r \in rets(\rho)$, each $\text{respond}_{i,j}(\rho, a, r)$ action is immediately preceded by an $\text{invoke}_{i,j}(\rho, a)$ action. We now prove our first simulation result.

Lemma 16: Let γ be an externally well-formed fair execution of System C. Then there exists a sequential fair execution β of System B such that for all $j \in \mathcal{J}, \beta|p_j = \gamma|p_j$.

Proof:[7] From the definition of System C, all processes p_j, $j \in \mathcal{J}$ and objects o_i, $i \in \mathcal{I}$ preserve well-formedness. Therefore, since γ is externally well-formed, we know that for all $j \in \mathcal{J}$, $\gamma|p_j$ is well-formed. Recall this means that:

- Every action in $\gamma_j|\mathcal{I}$ occurs from a prefix of γ_j after which p_j is active.
- The sequence $\gamma_j|\mathcal{I}$ is an alternating sequence of invoke and respond actions, beginning with an invoke action, such that $\forall i \in \mathcal{I}, \forall p \in ops(type(i)), \forall a \in args(p), \forall r \in rets(p)$, each $\text{respond}_{i,j}(p, a, r)$ action is immediately preceded in γ_j by an $\text{invoke}_{i,j}(p, a)$ action.

[7]This proof follows closely the proof of a similar theorem in [8] and uses ideas from [14], page 78, to treat the actions of the environment.

The second condition induces a total order $<_j$ on the operations invoked by p_j, and by Lemma 15, we know that each invocation has a matching response. Furthermore, since each object in System C is linearizable, we know that for each $i \in \mathcal{I}$, we can fix a fair execution γ_i of a_i such that for all $j, j' \in \mathcal{J}$,

1. $\gamma_i | p_j = (\gamma | o_i) | p_j$, and
2. if a respond$_{i,j'}$ event π' precedes an invoke$_{i,j}$ event π in γ, then π' precedes π in γ_i.

Each γ_i induces a total order $<_i$ on the operations invoked on o_i in γ.

In order to show that the execution β exists, we first show that there exists a total order $<_T$ on all the operations invoked in γ that is consistent with all the total orders $<_j, j \in \mathcal{J}$ and $<_i, i \in \mathcal{I}$. It is sufficient to show that the transitive closure $\prec$ of the union of all $<_j, j \in \mathcal{J}$ and $<_i, i \in \mathcal{I}$ is a partial order. Suppose not. Then there exists some cycle in $\prec$. We know that the cycle must involve a pair of operations ordered by $<_j$ for some $j \in \mathcal{J}$. Otherwise, all the operations in the cycle would be ordered by the *same* $<_i, i \in \mathcal{I}$, an immediate contradiction, since $<_i$ is a total order. Let op_1 and op_2 denote two operations ordered by $<_j$ in our cycle, and without loss of generality, let $op_1 <_j op_2$. This means that the response for op_1 occurs in γ before the invocation of op_2. Since op_1 and op_2 are in a cycle, we also know that there exists a sequence of operations $op_1, op_2, \ldots, op_n$ with $op_1 = op_n$ such that the response of each op_m precedes the invocation of op_{m+1}. But this means that the response of op_1 precedes the invocation of op_1 in γ, a contradiction.

So, to construct β, we first construct the schedule of β by taking the sequence of the invocation/response pairs in the order specified by $<_T$, and then, for all $j \in \mathcal{J}$, $k \in \mathcal{K}$, inserting in all *invoke*$_{j,k}$ and *respond*$_{j,k}$ actions appearing in γ so that $\beta | p_j = \gamma | p_j$. That is, we place each invocation/response pair of the environment "around" the corresponding sequence of object operations pairs, and place any invocation from the environment that is lacking a response after the last object operation pair. Now, we know from the construction of $<_T$ that for all $i \in \mathcal{I}$, $sched(\beta) | o_i = sched(\gamma) | o_i$. Furthermore, by the construction of $<_T$ and the alternating sequence condition of well-formedness for j, we know that for all $j \in \mathcal{J}$, $(sched(\beta) | \mathcal{I}) | p_j = (sched(\gamma) | \mathcal{I}) | p_j$. And from well-formedness, we know that for all $j \in \mathcal{J}$, every action in $\gamma_j | \mathcal{I}$ occurs from a prefix of γ_j after which p_j is active. Therefore, we know that it is possible to place the invocation/response pairs of the environment around the corresponding sequence invocation/response object operation pairs so that for all $j \in \mathcal{J}$, $sched(\beta) | p_j = sched(\gamma) | p_j$. Now, since all processes and objects have the same schedules in β as in γ, we can insert the states of β so that for all $j \in \mathcal{J}$, the sequence of state transitions in β for p_j is the same in β as in γ. (In other words, because its schedule is the same, each process p_j cannot tell whether it is in β or in γ.) Since each object o_i is a linearizable implementation of its sequential specification, we know that for all $i \in \mathcal{I}$, there exists a sequential fair execution β_i of a_i with schedule $sched(\gamma) | o_i$. Therefore, for each $i \in \mathcal{I}$, we let the sequence of state transitions of a_i in β be as in β_i. For each object a_i, we know that a response occurs in β for each invocation, so $\beta | a_i$ is fair. Since γ is fair and for all $j \in \mathcal{J}$, $\beta | p_j = \gamma | p_j$, we know that $\beta | p_j$ is fair, for all j. So, applying Lemma 1, we know that β is a sequential fair execution of System B. ∎

This result tells us that any externally well-formed fair execution of System C looks to the environment as if it is a sequential fair execution of System B. Now, we would like to say that any sequential fair execution of System B looks to the environment as if it is a fair execution of a system in which the objects are implemented in atomically accessed shared memory. This brings us to System A.

4.2.3 System A

In this section, we define System A, which forms the basis of our correctness condition. System A is a system in which objects are modelled as variables in a global shared memory that is accessed atomically by the processes. It is in the construction of System A (and the related proofs) that we exploit the shared memory extensions of the I/O automaton model. They allow us to model and reason about both the atomic access systems and the IR systems using a single unified model.

In order to define System A, we need a general transformation that takes a process automaton (as given to us in System C) and a set of sequential specifications (also given), and produces a shared memory automaton that corresponds to the original process but accesses the objects as atomic variables in a shared memory. We now define this transformation.

Given a process automaton p_j that accesses shared variables using the invocation-response mechanism as described above, we can construct an "equivalent" shared memory automaton s_j that accesses shared variables using atomic accesses to a shared memory. Since the transition relation of the shared memory automaton must specify how the shared variables are updated, the definition of s_j depends not only upon the definition of p_j, but also upon the sequential specifications for the objects that p_j accesses.

Changing the style of object access from invocation-response to atomic variable access is accomplished by a uniform syntactic transformation. We replace the invoke and respond actions in the signature by atomic shared memory actions. For all $i \in \mathcal{I}$, let X_i take on values from $states(i)$, and let $init(X_i) = init(i)$. Let $X = \bigcup_{i \in \mathcal{I}} X_i$, and $\bar{X}_i = X - \{X_i\}$. Automaton s_j is defined as follows.

- $sig(s_j)$ is the signature:
  Input Actions: $\text{invoke}_{j,k}(\rho, a)$, where $k \in \mathcal{K}, \rho \in ops(P_j)$ and $a \in args(\rho)$
  Output Actions: $\text{respond}_{j,k}(\rho, a, r)$, where $k \in \mathcal{K}, \rho \in ops(P_j)$, $a \in args(\rho)$, and $r \in rets(\rho)$
  $(v', \rho_{i,j}(a), v)$, where $v, v' \in dom(X)$, $i \in \mathcal{I}$, $\rho \in ops(type(i))$, and $a \in args(\rho)$.
- $states(s_j) = states(p_j)$,
- $start(s_j) = start(p_j)$,
- $steps(s_j)$ = the set of all steps (s'', π, s) such that either
  1. $\pi = \text{invoke}_{j,k}$ or $\text{respond}_{j,k}$, $k \in \mathcal{K}$ and $(s'', \pi, s) \in steps(p_i)$, or
  2. $\pi = (v', \rho_{i,j}(a), v)$ and $\exists r, s'$ such that
     (a) $(s'', \text{invoke}_{i,j}(\rho, a), s') \in steps(p_j)$,
     (b) $(s', \text{respond}_{i,j}(\rho, a, r), s) \in steps(p_j)$,
     (c) $P_\rho(a, v'|X_i) \Rightarrow Q_\rho(a, v'|X_i, v|X_i, r)$, and
     (d) $v|\bar{X}_i = v'|\bar{X}_i$.
- $part(s_j) = part(p_j)$, except that each $\text{invoke}_{i,j}(\rho, a)$ action is replaced by the actions $(v', \rho_{i,j}(a), v)$.

A few words explaining the transition relation for s_j are in order. We include directly in the steps of s_j each step of p_j for a process invocation or a response to the environment (i.e., each step not involving an object access). In addition, we include shared memory steps that correspond to invocation/response pairs for objects in System B. Conditions (a) and (b) say that the state change that occurs at s_j as a result of the atomic access corresponds to a state change that can occur in p_j as a result of the invocation/response pair. Condition (c) ensures that the new value of the shared variable X_i is consistent with the sequential specification S_i. Finally, condition (d) says that no shared variables other than X_i are changed by the step.

Lemma 17: For all $j \in \mathcal{J}, s_j$ is a shared memory automaton for X.
Proof: Immediate from the definitions of I/O automata and shared memory automata. ∎

We define System A to be the composition of the shared memory automata corresponding to the process automata of System C, closed out on the entire set of shared variables. More formally, $A = \mathcal{C}(\Pi_{j\in\mathcal{J}} s_j, X)$. We now show that for each execution of System B, there is an execution of System A in which the environment observes the same system behavior.

Lemma 18: Let β be a sequential fair execution of System B. Then there exists a fair execution α of System A such that $beh(\alpha)|\mathcal{K} = beh(\beta)|\mathcal{K}$.
Proof: We "collapse" β to get α: Since β is a sequential execution, each object operation invocation is immediately followed by its corresponding response. Therefore, to construct α, for all $i \in \mathcal{I}$, $j \in \mathcal{J}$, we replace each subsequence

$$s'', invoke_{i,j}(\rho, a), s', respond_{i,j}(\rho, a, r), s$$

in β by the corresponding step

$$(\hat{s''}, \rho_{i,j}(a), \hat{s}) \in steps(A)$$

in α such that for all $j' \in \mathcal{J}$ $\hat{s''}|s_{j'} = s''|p_{j'}$, $\hat{s}|s_{j'} = s|p_{j'}$, and for all $i' \in \mathcal{I}$, the values of $X_{i'}$ in $\hat{s''}$ and $\hat{s}$ match $(s''|a_{i'}).current$ and $(s|a_{i'}).current$, respectively. From the definition of a_i, we know that the state change at a_i is between s'' and s is consistent with the sequential specification S_i. Therefore, we know that the step $(\hat{s''}, \rho_{i,j}(a), \hat{s})$ must exist in *steps*(A). By the definition of s_j, α is an execution of A. To see that α is fair, we note that β is fair and each action $(v', \rho_{i,j}(a), v)$ of s_j is enabled exactly from those states in which invoke$_{i,j}(\rho, a)$ is enabled in p_j. ∎

Our main result follows immediately.

Theorem 19: Let γ be a fair execution of System C. Then there exists a fair execution α of System A such that $beh(\alpha)|\mathcal{K} = beh(\gamma)|\mathcal{K}$.
Proof: Immediate from Lemmas 16 and 18. ∎

Thus, we have shown formally that if the objects in an invocation-response system are each linearizable, then every fair behavior of the entire invocation-response system is a fair behavior of the corresponding atomic access system, as far as the components of the environment can tell.

5 Conclusion

We have extended the I/O automaton model to allow modelling of shared memory systems, as well as systems that include both shared memory and shared action communication. The extended model was shown to support all types of atomic accesses to shared memory, from the very restrictive single-variable reads and writes to operations on arbitrary abstract data types. By building our shared memory model on top of I/O automata, we could take advantage of existing features of the model, most notably composition and compositionality properties, fairness, and the separation of inputs and outputs. Using the built-in notion of an output action being under the control of a single process, we were able to capture the idea of a single module making an atomic update to shared memory (without any active participation by other modules). In addition, by exposing the values of the shared variables as part of the shared memory accesses, we were able to not only carry forward the compositionality properties of I/O automaton behaviors but also provide a useful notion of a shared memory action as an input.

We used the model to prove that invocation-response systems built from linearizable objects simulate atomic shared memory systems. This provided a demonstration that the I/O automaton model, when extended with our shared memory definitions, provides a unified framework in which we can prove relationships between message passing systems and shared memory systems.

Acknowledgments

We thank Nancy Lynch for her contributions to the definitions, and for her comments on earlier drafts.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The mit alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. Technical Report 694, Technion, Department of Computer Science, October 1991.

[3] Andrew Birrell, John Guttag, Jim Horning, and Roy Levin. Synchronization primitives for a multiprocessor: A formal specification. Technical Report 20, Digital Equipment Corporation Stanford Research Center, August 1987.

[4] K. Mani Chandy and Jayadev Misra. *A Foundation of Parallel Program Design*. Addison–Wesley, Reading, MA, 1988.

[5] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. Submitted for publication, 1993.

[6] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, 1991.

[7] Kenneth Goldman and Nancy Lynch. Modelling shared state in a shared action model. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[8] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th Principles of Programming Languages*, pages 13–26, January 1987. Also to appear in *Transactions on Programming Languages and Systems*.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690, September 1979.

[11] Leslie Lamport. On interprocess communication. Distributed Computing, 1(1):77–85,86–101, 1986.

[12] N. Lynch and M. Merritt. Introduction to the theory of nested transactions. In *International Conference on Database Theory*, pages 278–305, Rome, Italy, September 1986. Also, expanded version in Technical Report, MIT/LCS/TR-367, MIT Laboratory for Computer Science, July 1986. Revised version in *Theoretical Computer Science*, 62(1988):123-185.

[13] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of a distributed system. *Theoretical Computer Science*, 13:17–43, 1981.

[14] Nancy A. Lynch and Kenneth J. Goldman. Distributed algorithms. Technical Report MIT/LCS/RSS-5, MIT Laboratory for Computer Science, May 1989. MIT Research Seminar Series.

[15] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[16] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3), 1989.

[17] Bala Swaminathan and Kenneth J. Goldman. Hierarchical correctness proofs for recursive distributed algorithms using dynamic process creation. Technical Report WUCS-92-10, Washington University in St. Louis, September 1992. Revised April 1993.