LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-427

MODELLING SHARED STATE IN A SHARED ACTION MODEL

Kenneth Goldman
Nancy A. Lynch

March 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Modelling Shared State in a Shared Action Model

Kenneth J. Goldman and Nancy A. Lynch

March 15, 1990

Abstract

The I/O automaton model of Lynch and Tuttle is extended to allow modelling of shared memory systems, as well as systems that include 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 read-modify-write. The extended model supports system description, verification and analysis. As an example, Dijkstra's classical shared memory mutual exclusion algorithm is presented and proved correct.

Keywords: shared memory, formal models, distributed computing, distributed algorithms, mutual exclusion, input/output automata

This work 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.

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 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 [4], in which system components communicate by sending messages over synchronous channels, and UNITY [1], in which components communicate by reading and modifying shared variables.

The I/O automaton model [7, 8] 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. And in some cases one might wish to use both shared-memory and message-passing to describe different parts of an algorithm. Therefore, it would appear that introducing a shared-memory mechanism into the I/O automaton model would be a useful unification of these two approaches. The shared memory model of Lynch and Fischer [5] 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 extend 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. Each shared memory action contains extra 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). To illustrate these techniques, we present and prove the correctness of Dijkstra's classical shared memory mutual exclusion algorithm using the shared memory I/O automaton model.

The first author is currently developing the Spectrum Simulation System, a research tool for the design of distributed algorithms [3]. Spectrum consists of a language and simulator based on the I/O automaton model. Users express distributed algorithms as I/O automata and simulate them directly, using the semantics of the formal model. A graphical interface is provided for constructing systems of automata and animating their executions. Using I/O automaton composition, Spectrum users may define composed types hierarchically, study simulations at varying levels of detail, and create specialized debugging and analysis devices. Incorporating the shared memory extensions (specifically, the closeout operator) into this system will allow simulation of message-passing algorithms, shared memory algorithms, and hybrid algorithms all within a single formal framework. This is an added benefit of building a powerful unified model that accommodates both message-passing and shared memory communication. Although Spectrum does not yet support the closeout operator, we were able to use Spectrum to simulate the example algorithm presented in Section 4 by explicitly constructing the closed out automaton. The invariants and variant function were mechanically checked for random executions of the algorithm.

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. In Section 4, the extended model is used to present and prove correct Dijkstra's shared memory mutual exclusion algorithm. The paper concludes with a summary and discussion.

2 The I/O Automaton Model

The following introduction to the I/O automaton model is adapted from [8], which explains the model in more detail, presents examples, and includes comparisons to other models.

[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.

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. We will return to these points in Section 3.7.

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 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 $\vec{s_j}[i]$.

In defining the behaviors of a composition, it is sometimes convenient to *hide* actions, making them internal actions of the composition. The hidden actions are usually locally controlled actions of the composition that are also inputs to some of its own components.

[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 $\vec{s}$.

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.

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).

In our example progress proof of Dijkstra's mutual exclusion algorithm, we will rely on the built-in fairness feature of the I/O automaton model in order to reason about progress in a system containing several active, non-failing processes accessing passive shared memory.

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 $\Phi \cap int(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in out(M)$, and $\beta\pi|M \in finbehs(M)$. 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 $f'|(X \setminus \{x\}) = f|(X \setminus \{x\})$ 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[3].

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 $a \in \mathcal{L}$.[4] 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

[3] In some sense, this is the reverse of what is often done to incorporate message passing into a shared memory model. In UNITY [1], 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.

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

$\pi = (f'_X, a, f_X)$ 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 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 1: 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.

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.[5]

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:

- $in(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}in(A)\} \cup (in(A) \backslash shared\text{-}in(A))$.
- $out(B) = \{\pi \in sm\text{-}acts(Z) : \pi|X \in shared\text{-}out(A)\} \cup (out(A) \backslash shared\text{-}out(A))$.
- $int(B) = int(A)$.
- $states(B) = states(A)$.
- $start(B) = start(A)$.
- $steps(B) =$ 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.

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

[5] 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.

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 2: 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 5, says that augmentation does not (in any significant way) affect the behavior of an automaton. This is proved using the following lemmas.

Lemma 3: 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 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_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 5: 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 3 and 4.

For Part 2, let α_A be a fair execution of A, and let $\beta_A = beh(\alpha_A)$. From Lemma 3, 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 4, 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 6: 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 2, for each A_i, $augment(A_i, Z)$ is a shared memory automaton for Z. Therefore, by Theorem 1, the result holds. ∎

The following three compositionality results follow immediately from the corresponding results in [8], together with Theorems 5 and 6. The first result says that an execution of an augmented-composition induces executions of the component shared memory automata.

Corollary 7: 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 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 $A = \prod^+_{i \in I} A_i$. 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 $(\beta | augment(A_i, Z)) | X_i = sched(\alpha_i)$ for every $i \in I$. Then there is a (fair) execution α of A such that $\beta = sched(\alpha)$ 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 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 $A = \prod^+_{i \in I} A_i$. 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 $\{(s' \circ f_X,\ a,\ s \circ f_X) : f_X \in F_X\} \subseteq steps(B)$, 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 10: 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 11: 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 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_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 13: 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 11, 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 12 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 11 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 12 such that $\beta_A|(Z - X) = \mathit{beh}(\alpha_B)$. The remainder of the proof is argued as above. ∎

3.7 Discussion

Important in defining our shared memory extensions were the built-in features of the I/O automaton model, most notably composition and the separation of inputs and outputs. By 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 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).

4 Example: Dijkstra's mutual exclusion algorithm

In order to illustrate the shared memory extensions just presented, we apply them to Dijkstra's classical shared memory mutual exclusion algorithm. We begin by defining the mutual exclusion problem in terms of the I/O automaton model. We then present Dijkstra's algorithm as a composition of shared memory automata. The safety and progress proofs that follow demonstrate how proofs using standard assertional techniques may be expressed straightforwardly using this model.

4.1 The Mutual Exclusion Problem

Fix n, a positive integer, and let $\mathcal{I} = \{1, 2, \ldots, n\}$. We define schedule module M with $sig(\mathrm{M})$ as follows:

Inputs:	$\mathrm{UserTry}_i, i \in \mathcal{I}$	Outputs:	$\mathrm{Crit}_i, i \in \mathcal{I}$
	$\mathrm{UserExit}_i, i \in \mathcal{I}$		$\mathrm{Rem}_i, i \in \mathcal{I}$

Schedule module M interacts with an environment that may be thought of as a collection of n user processes u_i, $i \in \mathcal{I}$, where each process u_i has outputs $\mathrm{UserTry}_i$ and $\mathrm{UserExit}_i$, and has inputs Crit_i and Rem_i. A $\mathrm{UserTry}_i$ action means that process u_i wishes to enter its critical section. A Crit_i action by M gives u_i permission to enter its critical section. A $\mathrm{UserExit}_i$ action means that process u_i is leaving its critical section. Finally, the Rem_i action gives u_i permission to continue with the remainder of its program. If β is a sequence of actions of M, then we define $\beta|i$ to be the subsequence of β containing exactly the $\mathrm{UserTry}_i$, Crit_i, $\mathrm{UserExit}_i$, and Rem_i actions. Before defining the allowable schedules of M, we define the set of well-formed and user-live sequences of actions of M. Let β be a sequence of actions in $sig(\mathrm{M})$. We say that β is *well-formed* iff for all $i \in \mathcal{I}$, all prefixes of $\beta|i$ are prefixes of the infinite sequence $\mathrm{UserTry}_i$, Crit_i, $\mathrm{UserExit}_i$, Rem_i, $\mathrm{UserTry}_i$, $\mathrm{Crit}_i, \ldots$. This says, for example, that a process will not issue a try request while in its critical section. If β is a sequence of actions of S, we say that β is *user-live* iff for all $i \in \mathcal{I}$, $\beta|i$ is either infinite or does not end with Crit_i. Informally, this says that no user u_i stops in its critical section. An execution is said to be well-formed (user-live) iff its behavior is well-formed (user-live).

We define the set *scheds*(M), the allowable external behaviors of M, as follows. Let β be a sequence of actions in *sig*(M). Then $\beta \in scheds(M)$ iff the following conditions hold:

1. M preserves well-formedness in β.
2. If β is well-formed, then
   (a) (mutual exclusion) $\forall i, j \in \mathcal{I}$, if Crit_i and Crit_j occur in β (in that order), then UserExit_i occurs between them.
   (b) (progress) if β is user-live, then either β is infinite or $\forall i, \beta|i$ ends with Rem_i.

Condition (2a) is the safety property: no two processes are in their critical sections simultaneously. Condition (2b) is the progress property: either all processes eventually end up in their remainder regions or some process enters the critical region infinitely often. Both properties are guaranteed only if the user processes preserve well-formedness, and the progress condition is guaranteed only if user processes eventually exit the critical region. In this variant of the mutual exclusion problem, only a very weak progress requirement is made. For example, correct solutions to this problem admit executions in which a process is locked out of the critical section.

4.2 Dijkstra's Mutual Exclusion Algorithm

In this section, we model Dijkstra's shared memory mutual exclusion algorithm [2] as an illustration of our shared memory extensions to the I/O automaton model. As presented here, the variable names and structure more closely follow the description in [6], although the algorithm is the same.

We implement schedule module M by a collection of n automata p_i, $i \in \mathcal{I}$, where each p_i interacts with u_i through shared actions and interacts with the other p_j's using shared variables. Each p_i has three state components: *stage* $\in$ {try, read, check, set, control2, final_check, failed, crit, exit, done, remainder}; k, an integer in the range 1 to n; and *checked*, a set of integers in the range 1 to n. Initially, *stage* = remainder, k is arbitrary, and *checked* is the empty set. Automaton p_i is a shared memory automaton for V, where V has the following variables: k, an integer in the range 1 to n; and *control*[j] for $j \in \mathcal{I}$, which take on values from {0,1,2}. Initially, k has an arbitrary value and all *control* variables are 0. The code for automaton p_i is shown in Figure 1. Shared memory actions are listed by their access labels and distinguished by daggers (†); all other actions are listed by their action names. All actions of p_i are outputs, except UserTry_i and UserExit_i, which are its inputs. "Pre" and "Eff" denote "precondition" and "effect", respectively. For shared memory actions, the step $(s', (v', a, v), s)$ is in $steps(p_i)$ exactly when the precondition for a is satisfied in state s' and s and v are derived from s' and v' according to the effect clause. For all other output actions, the step (s', π, s) is in $steps(p_i)$ exactly when the precondition for π is satisfied in state s' and state s is derived from state s' according to the effect clause. If an action has no precondition, it is always enabled. If a state component or variable is not mentioned in the effect clause, it is left unchanged by the action. The partition consists of a class for each $i \in \mathcal{I}$ that contains all the output actions of p_i.

Essentially, the algorithm proceeds in two stages. After receiving a UserTry_i input, p_i sets its control variable to 1 and enters stage one. In stage one, it continually reads k and checks to see if control[k] is 0. If it finds a 0 in control[k], it sets k to its own index i. If it reads k and finds it equal to i, p_i proceeds to stage two and sets its control variable to 2. In stage two, p_i performs a final check by examining the control variables of all the other processes. If any of these control variables are found to be 2, then p_i fails and returns to stage one (where it sets its control variable back 1).

- UserTry_i
  Eff: $s.\text{stage}_i = \text{try}$

† Try_i
  Pre: $s'.\text{stage}_i \in \{\text{try}, \text{failed}\}$
  Eff: $v.\text{control}[i] = 1$
  $s.\text{checked}_i = \{i\}$
  $s.\text{stage}_i = \text{read}$

† Read_i
  Pre: $s'.\text{stage}_i = \text{read}$
  Eff: $s.k_i = v'.k$
  if $s.k_i = i$ then
  $s.\text{stage}_i = \text{control2}$
  else
  $s.\text{stage}_i = \text{check}$

† $\text{Check}(j)_i$
  Pre: $s'.\text{stage}_i = \text{check}$
  $j = s'.k_i$
  Eff: if $v'.\text{control}[j] = 0$ then
  $s.\text{stage}_i = \text{set}$
  else
  $s.\text{stage}_i = \text{read}$

† Set_i
  Pre: $s'.\text{stage}_i = \text{set}$
  Eff: $v.k = i$
  $s.\text{stage}_i = \text{read}$

† Control2_i
  Pre: $s'.\text{stage}_i = \text{control2}$
  Eff: $v.\text{control}[i] = 2$
  $s.\text{stage}_i = \text{final\_check}$

† $\text{FinalCheck}(j)_i$
  Pre: $s'.\text{stage}_i = \text{final\_check}$
  $j \notin s'.\text{checked}_i$
  Eff: if $v'.\text{control}[j] = 2$ then
  $s.\text{stage}_i = \text{failed}$
  else
  $s.\text{checked}_i = s'.\text{checked}_i \cup \{j\}$

- Crit_i
  Pre: $s'.\text{stage}_i = \text{final\_check}$
  $|s'.\text{checked}_i| = n$
  Eff: $s.\text{stage}_i = \text{crit}$

- UserExit_i
  Eff: $s.\text{stage}_i = \text{exit}$
  $s.\text{checked}_i = \{i\}$

† Reset_i
  Pre: $s'.\text{stage}_i = \text{exit}$
  Eff: $v.\text{control[i]} = 0$
  $s.\text{stage}_i = \text{done}$

- Rem_i
  Pre: $s'.\text{stage}_i = \text{done}$
  Eff: $s.\text{stage}_i = \text{remainder}$

Figure 1: Transition Relation for p_i in Dijkstra's Algorithm

Otherwise, p_i finds all the control variables to be less than 2 and issues a Crit_i action, allowing u_i to proceed to the critical section. After u_i leaves the critical section (and issues a UserExit_i action), p_i resets its control variable to 0 and issues a Rem_i action.

We associate with each p_i an access partition ψ^i as follows: For each $j \in \mathcal{I}$, $\psi^i_r(control[j])$ contains the labels $\text{Check}(j)_i$ and $\text{FinalCheck}(j)_i$. Also, $\psi^i_w(control[i])$ contains the labels Try_i, Control2_i, and Reset_i. And for each $j \neq i$, $\psi^i_w(control[j])$ is empty. Finally, $\psi^i_r(k)$ contains Read_i and $\psi^i_w(k)$ contains Set_i. The following result follows immediately from inspection of the code.

Lemma 14: For all $i \in \mathcal{I}$, automaton p_i is a single-variable read-write automaton under ψ^i.

We let system $S = \mathcal{C}(\Pi_{1 \leq i \leq n} p_i, \{k, control[i], i \in \mathcal{I}\})$ be the composition of the processes of Dijkstra's algorithm, closed out on k and the *control* variables. Furthermore, we hide all shared memory actions of S so that the external signatures of M and S are the same. One may note that all the p_i's in system S can read and write shared variable k, whereas the variable control[i] may be written only by p_i and read by the other p_j's. That is, each *control*[i] is owned by p_i, while k is a multi-writer variable.

We wish to show that system S solves schedule module M. The proof has three parts. First, we show that S preserves well-formedness in all executions, Condition (1) of module M. In Section 4.3, we give the safety proof, Condition (2a). Finally, we present the progress proof, Condition (2b), in Section 4.4.

If i is a process index and s is a state of system S, we say that process p_i is a *contender* in state s, written $contender(i, s)$, iff $s.\text{stage}_i \in \{\text{read, check, set, control2, final\_check, failed}\}$.

Lemma 15: Let α be an execution of system S with behavior β. Then system S preserves well-formedness in β.

Proof: By induction on the length of α. For the base case, if α contains no actions, then it is well-formed. Let $\alpha = \alpha' s \pi$, where $beh(\alpha')$ is well-formed and π is an output action of S. There are two cases.

- If π is a Crit_i action, then by the preconditions of that action it must be that p_i is a contender in state s. Therefore, the last action in $beh(\alpha')|i$ must be UserTry_i, for any other action would leave p_i in a non-contender state.

- If π is a Rem_i action, then by the preconditions of that action it must be that $\text{stage}_i = \text{done}$ in state s. Therefore, the last action in $\beta'|i$ must be Reset_i, for any other action would leave p_i in a state with $\text{stage}_i \neq \text{done}$. Since Reset_i is only enabled when $\text{stage}_i = \text{exit}$, the last action in $beh(\beta')|i$ must be UserExit_i, for any other action would leave p_i in a state with $\text{stage}_i \neq \text{exit}$.

In both cases, β is well-formed. ∎

The following lemma will be used in the safety proof to rule out the occurrence of UserTry and UserExit actions from certain states.

Lemma 16: Let α be an execution of system S with behavior β. If β is well-formed, then for all states s in α, if s is immediately followed by a UserTry_i (UserExit_i) action, then $s.\text{stage}_i$ is remainder (crit).

Proof: If s is followed by UserTry_i, then by definition of well-formedness, the preceding action in $\beta|i$ is a Rem_i action, and a Rem_i action leaves $\text{stage}_i = \text{remainder}$. Furthermore, no output actions of p_i are enabled while $\text{stage}_i = \text{remainder}$. If s is followed by UserExit_i, then by definition of well-formedness, the preceding action in $\beta|i$ is a Crit_i action, and a Crit_i action leaves $\text{stage}_i = \text{crit}$. Furthermore, no output actions of p_i are enabled while $\text{stage}_i = \text{crit}$. ∎

4.3 Safety Proof

Let s be a state of system S. To denote the set of processes in (or about to enter) their critical sections, we define the set $ready(s) = \{i : (s.\text{stage}_i = \text{crit}) \vee (|s.\text{checked}_i| = n)\}$. The proof is based on a set of invariants, proved in the following Lemma.[6] Using Spectrum, this Lemma was checked mechanically for all states of random executions of the algorithm.

Lemma 17: Let α be a well-formed execution of system S. In states s of α, for all processes p_i and p_j, the following facts hold:

1. $s.\text{control}[i] = 2$ iff $s.\text{stage}_i \in \{\text{final\_check, failed, crit, exit}\}$.
2. If $s.\text{checked}_i \neq \{i\}$ then $s.\text{stage}_i \in \{\text{final\_check, failed, crit}\}$.
3. If $i \neq j$, then $i \in s.\text{checked}_j \Rightarrow j \notin s.\text{checked}_i$.
4. If $i \in \text{ready}(s)$ then $s.\text{checked}_i = \{1, 2, ...n\}$.
5. If $s.\text{stage}_i \in \{\text{control2, final\_check, failed, crit, exit, done}\}$, then $s.k_i = i$.

Proof: In the initial state of S, $\forall i \in \mathcal{I}$, control$[i] = 0$, checked$_i = \{i\}$, and stage$_i$ = remainder. Therefore, all the facts hold in the initial state. Let $\alpha = \alpha'\pi s$, and assume that the facts hold in all states of α', and specifically in the last state s' of α'. We consider each fact in turn, showing that it must hold in state s as well.

1: If $s'.\text{control}[i] = 2$, then by the induction hypothesis $s'.\text{stage}_i \in S = \{\text{final\_check, failed, crit, exit}\}$. Therefore, π must be either Try$_i$, FinalCheck$(j)_i$, Crit$_i$, UserExit$_i$, or Reset$_i$. (Lemma 16 rules out UserTry$_i$.) The actions FinalCheck, Crit$_i$, and UserExit$_i$ do not change the value of control$[i]$ and result in $s.\text{stage}_i \in S$. The actions Try$_i$ and Reset$_i$ both cause $s.\text{control}[i] \neq 2$, but also result in $s.\text{stage}_i \notin S$. Therefore, the property is preserved if $s'.\text{control}[i] = 2$.

If $s'.\text{control}[i] \neq 2$, then by the induction hypothesis $s'.\text{stage}_i \notin S$. Therefore, π must be either UserTry$_i$, Try$_i$, Read$_i$, Check$(j)_i$, Set$_i$, Control2$_i$, or Rem$_i$. (By Lemma 16, UserExit$_i$ is ruled out.) Actions UserTry$_i$, Read$_i$, Check$(j)_i$, Set$_i$, and Rem$_i$ do not change the value of control$[i]$ and result in $s.\text{stage}_i \notin S$. Furthermore, the action Try$_i$ sets control$[i] = 1$ and results in $s.\text{stage}_i \notin S$. Finally, the action Control2$_i$ sets control$[i] = 2$, but also results in $s.\text{stage}_i \in S$. Therefore, the property is preserved if $s'.\text{control}[i] \neq 2$.

2: If $s'.\text{checked}_i = \{i\}$, then the only possibility for π which could cause $s'.\text{checked}_i \neq \{i\}$ is FinalCheck$(j)_i$. This action is only enabled if $s'.\text{stage}[i] = \text{final\_check}$. The FinalCheck$(j)_i$ either does not change stage$_i$ or sets $s.\text{stage}_i = \text{failed}$. Therefore, the property is preserved.

If $s'.\text{checked}_i \neq \{i\}$, then by the induction hypothesis, $s'.\text{stage}_i \in \{\text{final\_check, failed, crit}\}$. Therefore, the only possibilities for π which could cause $s.\text{stage}_i \notin \{\text{final\_check, failed, crit}\}$ are Try$_i$ and UserExit$_i$. (The action UserTry$_i$ is ruled out by Lemma 16.) However, in both cases, $s.\text{checked}_i = \{i\}$, so the property is preserved.

3: The proof is by contradiction. Suppose $\exists i \neq j$ such that $i \in s.\text{checked}_j$ and $j \in s.\text{checked}_i$. Without loss of generality, suppose that $i \in s'.\text{checked}_j$, and let π be the action that adds j to checked$_i$. (By the induction hypothesis, we know that $j \notin s'.\text{checked}_i$.) The only possibility

[6] Although the last invariant of Lemma 17 is used only in the liveness proof, we present it here because of its similarity to the others.

for π is FinalCheck$(j)_i$. By the transition relation, π can only add j to checked$_i$ if s'.control$[j]$ $\neq 2$. However, by the induction hypothesis (Fact 2), we know that s'.stage$_j \in$ {final_check, failed, crit}, since s'.checked$_j \neq \{j\}$. Therefore, by Fact 1, we know that s'.control$[j] = 2$, a contradiction.

4: Recall, from the definition, that $i \in$ ready(s) iff s.stage$_i$ = crit $\vee$ $|s$.checked$_i| = n$. By a pigeonhole argument, the fact clearly holds when $|s$.checked$_i| = n$. If s'.stage$_i \neq$ crit, then the only possibility for π to make s.stage$_i$ = crit is the Crit$_i$ action. That action has as a precondition that $|$checked$_i| = n$, and does not change the value of checked$_i$. Therefore, the property is preserved. If s'.stage$_i$ = crit, then the only possibility for π to make $|s$.checked$_i| \neq$ n is UserExit$_i$, but this also results in stage$_i$ = exit.

5: If s'.stage$_i \in$ {control2, final_check, failed, crit, exit, done}, then by the inductive hypothesis, $s'.k_i = i$. Furthermore, the only action which can change k_i is a Read$_i$ action, which is only enabled if stage$_i$ = read, so $s.k_i = s'.k_i = i$. If s'.stage$_i \notin$ {control2, final_check, failed, crit, exit, done}, then the only possible action for π which could cause s.stage$_i$ to be in that set is a Read$_i$ action. (Lemma 16 rules out UserExit$_i$.) However, the Read$_i$ action can only set s.stage$_i$ = control2 if $s.k_i = i$. Thus, the property is preserved.

All five facts hold in state s. ∎
We can now show that no two processes may be in (or about to enter) their critical sections.

Theorem 18: If s is a state of system S, $|$ready$(s)| \leq 1$.

Proof: By contradiction. Suppose that $|$ready$(s)| > 1$. Then by Fact 4 of Lemma 17, there must exist two processes p_i and p_j such that s.checked$_i$ = s.checked$_j$ = $\{1, 2, ...n\}$. However, this contradicts Fact 3 of Lemma 17. ∎

It follows that the algorithm satisfies mutual exclusion.

Corollary 19: Let α be a well-formed execution of system S. Then $\forall i, j \in \mathcal{I}$, if Crit$_i$ and Crit$_j$ occur in α (in that order), then UserExit$_i$ occurs between them.

Proof: By well-formedness and inspection of the code for system S, if a Crit$_i$ action occurs in α then stage$_i$ = crit in all states up until the next UserExit$_i$ action. Suppose (for contradiction) that there exist two processes p_i and p_j such that Crit$_i$ and Crit$_j$ occur in α (in that order) and no UserExit$_i$ occurs between them. Then after Crit$_j$ occurs, stage$_i$ = crit and stage$_j$ = crit. However, by Theorem 18 and the definition of *ready*, this is impossible. ∎

4.4 Progress Proof

In this section, we show that Dijkstra's algorithm makes progress: if a process is attempting to enter the critical section, then eventually it or some other process will enter the critical section. We define a "no-progress execution" of system S and then show that no such executions exist. The proof is by contradiction: We define a well-founded variant function, or progress metric. Then we show that in no-progress executions the function is nonincreasing and must eventually decrease. Since no infinite-length decreasing chains are possible, this shows that no-progress executions do not exist. The notion of fairness, which we inherit "for free" from the original I/O automaton model, is used to show that the variant function eventually decreases.

Let $\gamma = \alpha\beta$ be a fair well-formed user-live execution of system S. Furthermore, let none of the following actions occur in β: UserTry, Crit, UserExit, Rem. If β begins with a state in which some process has stage $\neq$ remainder, then β is said to be a *no-progress execution suffix* and γ is said to be a *no-progress execution.*

Lemma 20: Let β be a no-progress execution suffix, and let s be a state in β. Then $\forall i \in \mathcal{I}$, $s.\text{stage}_i \notin \{\text{crit, exit, done}\}$.

Proof: Immediate from the definitions of no-progress execution suffix, fairness, and p_i. ∎

Before defining the variant function, we identify an important predicate on system states. If s is a state of S, we say that s is *consistent*, denoted *consistent*(s), iff for all $i \in \mathcal{I}$, contender(i, s) $\Rightarrow s.k_i = s.k$.

We now define the variant function. Given state s of system S, we define

$$f(s) = (A, B, C, D, E, F, G, H, I, J, K),$$

where each tuple component has the nonnegative integer value defined as follows:

$A =$ $|\{i : s.\text{stage}_i = \text{try}\}|$.

$B =$ $|\{i : s.\text{stage}_i = \text{read}\}|$ if $\neg$contender$(s.k, s)$,
0 otherwise.

$C =$ $|\{i : s.\text{stage}_i = \text{check} \wedge \neg\text{contender}(s.k_i, s)\}|$.

$D =$ 0 if contender$(s.k, s)$, 1 otherwise.

$E =$ $|\{i : s.\text{stage}_i = \text{set}\}|$.

$F =$ $|\{i : s.\text{stage}_i = \text{control2} \wedge i \neq s.k\}|$.

$G =$ $|\{i : s.\text{stage}_i = \text{final\_check} \wedge i \neq s.k\}|$.

$H =$ $|\{i : \text{contender}(i, s) \wedge k_i \neq s.k\}|$.

$I =$ $\Sigma_i(n - |\text{checked}_i|)$, for all $i \neq s.k$ such that
$s.\text{stage}_i = \text{final\_check}$.

$J =$ $|\{i : s.\text{stage}_i = \text{failed} \wedge i \neq s.k\}|$.

$K =$ n if $(\neg\text{consistent}(s) \vee s.\text{stage}_{s.k} \neq \text{final\_check})$,
$n - |\text{checked}_{s.k}|$ otherwise.

We define a lexicographic total order on the elements in the range of f. We will show that f is nonincreasing and will eventually decrease in a no-progress execution suffix, but first we explain the components of f. The components A, E, F, G, and J simply count the number of processes in a certain stage (in some cases ignoring the process whose index is the value of the shared variable k). These components measure progress of the contenders through the different stages of the algorithm. Components B and C serve a similar purpose for the "read" and "check" stages, but are more complicated because contenders may cycle through these two stages while they wait for some other process to "get out of the way." Component B's purpose is to count the number of processes in the "read" stage; however, when the shared variable k is the index of a contender, $B = 0$. In this way, the value of B does not increase when a contender "backs off" to read k again. Component C counts the number of processes in the "check" stage whose local k variables contain indices of non-contenders.

Component D becomes 0 when the shared variable k is set to the index of a contender, and remains 1 otherwise. Components I and K both count down the number of indices that are missing

from the *checked* sets of processes whose stage is "final_check." Component I hold the sum of this count for all the contenders whose indices are not equal to the shared variable k. Component K holds this count for the (at most one) contender whose index is equal to the shared variable k, but only starts counting down after all other contenders are "out of the way," meaning that their local k's are equal to the shared k.

In studying the variant function, two important progress "landmarks" should be noted. The first is when component D reaches 0, after which point the value of k is always the index of a contender. After D reaches 0, the second landmark is when component H reaches 0, meaning that all later states are consistent. After this point, all processes other than p_k cannot escape the Read-Check cycle, so nothing stands in p_k's way.

We now show that the value of the variant function f is nonincreasing in no-progress execution suffixes, and that only certain steps leave f unchanged. Using Spectrum, this lemma (and earlier incorrect versions of it) was checked for random algorithm executions. That is, for each step it was mechanically verified that either (1) progress was being made (see Lemma 20), or (2) the variant function decreased, or (3) the variant function was unchanged and one of the exceptions held.

Lemma 21: Consider any state s' in β, a no-progress execution suffix. If action π of process p_i occurs from state s' producing state s, then the following conditions hold:

1. $f(s') \geq f(s)$, and
2. either $f(s') > f(s)$ or one of the following hold:
   (a) π is a Read action and $s'.k_i = s'.k$, a contender, or
   (b) π is a Check action and $s'.k_i$ is a contender, or
   (c) π is a Try action, $i = s'.k$, and $s'.\text{stage}_i$ = failed, or
   (d) π is a Control2 action and $i = s'.k$, or
   (e) π is a FinalCheck action, $i = s'.k$, and $\neg\text{consistent}(s')$.

Condition (1) says that the variant function is nonincreasing. Condition (2) says that every action must decrease the variant function, except for a few special cases. Exceptions (2a) and (2b) handle the case of a process cycling through the "read" and "check" stages, waiting for some other process to get out of the way. Note the relationship between items (2a) and (2b) and the variant function components B and C, respectively. A process does not make progress when it reads the same value of the shared variable k that it read the previous time. Similarly, a process does not make progress if it discovers that the control variable corresponding to its local k is nonzero. Exceptions (2c), (2d), and (2e) pertain only to the contender whose index is the value of the shared variable k. Process p_k may "back off" several times before it finally enters the critical section, and the variant function is carefully constructed not to change when p_k backs off. These last three exceptions are the necessary result.

Proof: By case analysis. For each possible action, we note the changes in the components of the variant function f. (We will use A' and A to denote the first components of $f(s')$ and f(s), respectively. Similarly for B' and B, etc.) Each case may be verified by Lemma 20 and inspection of the preconditions and effects of the action under consideration.

- If $\pi = (v', \text{Try}_i, v)$, there are three cases:
  (1) If $s'.\text{stage}_i$ = try, then $A' > A$, decreasing f.

(2) If $s'.\text{stage}_i = \text{failed}$ and $i \neq s'.k$, then $J' > J$, and no components increase. (Component B cannot increase because Fact (5) from Lemma 17 tells us that if $\text{stage}_i = \text{failed}$, then $k_i = i$, a contender by definition.) Therefore, f is decreased.
(3) If $s'.\text{stage}_i = \text{failed}$ and $i = s'.k$, then $f(s') = f(s)$, satisfying Condition 1 and exception 2c.

- If $\pi = (v', \text{Read}_i, v)$, there are three cases:
(1) If $s'.k$ is not a contender, then $B' > B$ and A is unchanged, so f decreases.
(2) If $s'.k_i \neq s'.k$, then $H' > H$ and no higher order components are increased, so f decreases.
(3) If $s'.k_i = s'.k$, a contender, then $f(s') = f(s)$, satisfying Condition 1 and exception 2a.

- If $\pi = (v', \text{Check}(j)_i, v)$, there are two cases:
(1) If $s'.k_i$ is a contender, then $f(s') = f(s)$, satisfying Condition 1 and exception 2b.
(2) Otherwise, $C' > C$, and A and B are unchanged, so f is decreased.

- If $\pi = (v', \text{Set}, v)$, then $B = 0$, $D = 0$, $E' > E$, and A and C are unchanged. Therefore f decreases.

- If $\pi = (v', \text{Control2}_i, v)$, there are two cases:
(1) If $i = s'.k$ then $f(s') = f(s)$, satisfying Condition 1 and exception 2d.
(2) Otherwise, $F' > F$ and no higher order components are changed, so f decreases.

- If $\pi = (v', \text{FinalCheck}(j)_i, v)$, there are three cases:
(1) If $i \neq s'.k$, then $I' > I$ and no higher order components are changed, so f decreases.
(2) If $i = s'.k$ and $\neg\text{consistent}(s')$, then $f(s') = f(s)$, so Condition 1 and exception 2d are satisfied.
(3) If $i = s'.k$ and $\text{consistent}(s')$, then K is the only component that may change. Suppose, for contradiction, that K does not decrease. By the effects of FinalCheck and the definition of K, the only way for this to happen is for $s'.\text{control}[j] = 2$. If $s'.\text{control}[j] = 2$, then Fact 1 of Lemma 17 tells us that $s'.\text{stage}[j] \in \{\text{final\_check}, \text{failed}, \text{crit}, \text{exit}\}$. Therefore, by Fact 5 of the same Lemma, $s'.k_j = j$. Since s' is consistent, $s'.k_j = s'.k$, and we have stated that $s'.k = i$. So, by transitivity, $j = i$. By the preconditions on FinalCheck, $j \notin s'.\text{checked}_i$. But $i \in s'.\text{checked}_i$, since $i \in \text{checked}_i$ initially and no action may remove it from that set. Therefore $j \neq i$, a contradiction.

In each case, the Lemma holds. The set of cases is complete by Lemma 20 and the definition of a no-progress execution. ∎

We have just shown that the value of the variant function f never increases in a no-progress execution suffix, and that only certain steps leave its value unchanged. Now we will show that a fair execution cannot proceed using only those certain steps, so the function must eventually decrease.

Corollary 22: Let α be a no-progress execution suffix. Then f must eventually decrease in α.

Proof: Suppose that f is fixed in α', a suffix of α. Then, by Lemma 21 for all states s' of α', if π occurs from s', then one of the following hold:

- π is a Read action and $s'.k_i = s'.k$, a contender, or

- π is a Check action and $s'.k_i$ is a contender, or

- π is a Try action, $i = s'.k$, and $s'.\text{stage}_i = \text{failed}$, or

- π is a Control2 action and $i = s'.k$, and
- π is a FinalCheck action, $i = s'.k$, and $\neg$consistent(s').

Since no action in α' is a Set action, the shared variable k is fixed in α'. *Fairness tells us that all contenders must continue taking steps.* (Inspection of the code will reveal that a contender always has *some* step enabled.) Therefore, by the four conditions above, all contenders other than p_k must have stage $\in$ {read, check}; otherwise their steps would decrease the value of f, contradicting our assumption that f is fixed. Therefore, by the same fairness argument, a Read action must eventually occur for each of these contenders, after which point its local value of k matches the shared k.

Let α'' be the suffix of α' after which all contenders other than p_k have their local k's equal to the shared k. Now, consider p_k, which must continue to take steps in α'', and let s'' be a state in α'' from which p_k takes a step. If p_k takes a FinalCheck step, then by Fact 5 of Lemma 17, $s''.k_k = s''.k$. However, this implies that s'' is consistent. Therefore, the conditions above imply that no FinalCheck actions can occur. If p_k takes a Control2 step, then a FinalCheck action will become enabled and remain enabled until it occurs, so fairness tells us that a FinalCheck action will eventually occur, but we have just ruled this out. The only remaining actions for p_k are Read, Check, and Try. If p_k takes a Read step, then it will observe that the shared k contains its own index and proceed to stage = control2, meaning that it must eventually take a Control2 step, which we have already ruled out. If p_k takes a Check step, then since (by statement 2 above) $s''.k_k$ is a contender, it will proceed to stage = read, meaning that it must eventually take a Read step, which we have just ruled out. Finally, if p_k takes a Try step, it will also proceed to stage = read. Therefore, if p_k continues to take steps, it eventually will decrease the value of f, giving us our contradiction. ∎

Our main liveness result follows immediately.

Theorem 23: The set of no-progress executions for Dijkstra's algorithm is empty.

Proof: By Lemma 21, we know that the value of the variant function f is nonincreasing in a no-progress execution suffix. Furthermore, by Lemma 22, the value of f never reaches a fixed point. Therefore, since f cannot infinitely decrease, the theorem holds. ∎

Finally, we show that the above theorem implies that Dijkstra's algorithm satisfies the required progress property.

Corollary 24: Let α be a fair well-formed user-live execution of system B. Then either $\forall i, \alpha|i$ ends with Rem_i, or $\exists i$ such that $\alpha|i$ is infinite.

Proof: By contradiction. Suppose that α is finite and that there exists some $l \in \mathcal{I}$ such that $\alpha|l$ does not end with Rem_i. Then there exists a suffix of α in which p_l has stage $\neq$ remainder and $\alpha|i$ is empty for all i. This is a no-progress execution suffix, by definition. Therefore α is a no-progress execution, which is a contradiction of Theorem 23. ∎

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 the fairness definitions and compositionality properties already present in that model. This resulted in a unified model with relatively few new concepts. An example algorithm, Dijkstra's classical shared memory mutual exclusion algorithm, was presented in this model and its safety and progress properties were shown using standard assertional and variant function techniques.

Acknowledgment

We thank Kathy Yelick for her careful reading of an earlier draft.

References

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

[2] E.W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[3] Kenneth J. Goldman. Distributed algorithm simulation using input/output automata. Ph.D. Thesis, M.I.T. Laboratory for Computer Science, in progress.

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

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

[6] Nancy A. Lynch and Kenneth J. Goldman. *Distributed Algorithms.* Technical Report MIT/LCS/RSS-5, M.I.T. Laboratory for Compuer Science, May 1989. MIT Research Seminar Series.

[7] 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.

[8] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.