

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering
Research

Computer Science and Engineering

Report Number: WUCS-91-43

1991-07-01

Composition, Superposition, and Encapsulation in the Formal Specification of Distributed Systems

Kenneth J. Goldman

Composition, superposition, and encapsulation are important techniques that work well together for designing large distributed software systems. Composition is a symmetric operator that allows system components to communicate with each other across module boundaries. Superposition is an asymmetric relationship that allows one system component to observe the state of another. Encapsulation is the ability to define the reason about the behavior of a module in terms of a well-defined boundary between that module and its environment, while hiding the internal operations of that module. In this paper, the I/O automation model of Lynch and Tuttle is extended to permit... [Read complete abstract on page 2.](#)

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Goldman, Kenneth J., "Composition, Superposition, and Encapsulation in the Formal Specification of Distributed Systems" Report Number: WUCS-91-43 (1991). *All Computer Science and Engineering Research*.

https://openscholarship.wustl.edu/cse_research/661

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Composition, Superposition, and Encapsulation in the Formal Specification of Distributed Systems

Kenneth J. Goldman

Complete Abstract:

Composition, superposition, and encapsulation are important techniques that work well together for designing large distributed software systems. Composition is a symmetric operator that allows system components to communicate with each other across module boundaries. Superposition is an asymmetric relationship that allows one system component to observe the state of another. Encapsulation is the ability to define the reason about the behavior of a module in terms of a well-defined boundary between that module and its environment, while hiding the internal operations of that module. In this paper, the I/O automation model of Lynch and Tuttle is extended to permit superposition of program modules. This results in a unified model that supports composition, superposition, and encapsulation. The extended model includes a formal specification mechanism for layered systems that allows the sets of correct behaviors of each layer to be expressed in terms of the states of the layers below it. To illustrate the ideas, we use the extended model to specify the global snapshot problem and prove the correctness of the global snapshot algorithm of Chandy and Lamport.

Composition, Superposition, and Encapsulation in
the Formal Specification of Distributed Systems

Kenneth J. Goldman

WUCS-91-43

July 1991, revised April 1992
previously, "A Compositional Model for Layered Distributed Systems"

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

Composition, Superposition, and Encapsulation in the Formal Specification of Distributed Systems ¹

Kenneth J. Goldman ²

Department of Computer Science
Washington University
St. Louis, MO 63130-4899
kjpg@cs.wustl.edu

April 1, 1992

¹Portions of this work have appeared elsewhere in preliminary form [8].

²Part of this research was conducted at the Massachusetts Institute of Technology Laboratory for Computer Science 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, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and part of this research was conducted at Washington University and supported in part by the National Science Foundation under Grant CCR-91-10029.

Abstract

Composition, superposition, and encapsulation are important techniques that work well together for designing large distributed software systems. Composition is a symmetric operator that allows system components to communicate with each other across module boundaries. Superposition is an asymmetric relationship that allows one system component to observe the state of another. Encapsulation is the ability to define and reason about the behavior of a module in terms of a well-defined boundary between that module and its environment, while hiding the internal operations of that module.

In this paper, the I/O automaton model of Lynch and Tuttle is extended to permit superposition of program modules. This results in a unified model that supports composition, superposition, and encapsulation. The extended model includes a formal specification mechanism for layered systems that allows the set of correct behaviors of each layer to be expressed in terms of the states of the layers below it. To illustrate the ideas, we use the extended model to specify the global snapshot problem and prove the correctness of the global snapshot algorithm of Chandy and Lamport.

Keywords: composition, compositionality, distributed algorithms, encapsulation, formal models, global snapshot, I/O automata, layered systems, specification, superposition

1 Introduction

The *structure* of a system design is often the single most important factor in the success of a design, because the details of the system implementation must fit within that structure. It is important that the structure of a design be “natural” so that the details “fall into place.” This notion applies at all levels, from the design of intricate distributed algorithms to the design of large heterogeneous systems. Just as the mechanisms available in a programming language for structuring a program can have a great impact on the way one thinks about writing software in that language, the mechanisms available for specifying a problem or an algorithm in a formal model can influence the way in which one thinks about the problem itself. Although a case can be made for keeping models as simple as possible, it is important that a formal model support a range of structuring techniques so that different aspects of a problem can be modeled naturally and appropriately.

In this paper, we are concerned with ways to support structured design of distributed systems within formal models. Specifically, we are interested in ways to support composition, superposition, and encapsulation within a single unified model. Composition, superposition, and encapsulation are important techniques that work well together for designing large distributed software systems. Composition is a symmetric operator that allows system components to communicate with each other across module boundaries. Superposition is an asymmetric relationship that allows one system component to observe the state of another; it permits modular descriptions of distributed algorithms in terms of several program layers, in which higher layers are allowed to make use of lower layers, but lower layers are unaware of the higher layers. Encapsulation is the ability to define and reason about the behavior of a module in terms of a well-defined boundary between that module and its environment, while hiding the internal operations of that module.

Our approach to developing a unified model that supports composition, superposition, and encapsulation is to start with two existing models, each supporting only some of these features, and then to develop extensions to one of the models to support the features of the other. The two models we study are UNITY [3], which provides a superposition operator, and I/O Automata [13], which provides composition and encapsulation. The result of this work is an extended I/O automaton model that supports all three of these features and includes a formal specification mechanism for layered systems that allows the set of correct behaviors of each layer to be expressed in terms of the states of the layers below it.

The I/O automaton model of Lynch and Tuttle [14] is particularly natural for describing distributed systems. It permits the writing of precise problem specifications, clear algorithm descriptions, and careful correctness proofs. In this model, complex systems are described as the composition of simpler system components (modules). Communication takes place entirely in terms of actions shared across module boundaries. Each module has its own local state variables, unseen by other modules. This encapsulation, which gives rise to the compositionality results of the model, makes it possible to reason locally about system components in order to prove properties about executions of the entire system. However, the I/O automaton model does not provide a mechanism for constructing layered systems in which higher level modules can observe the states of lower level ones.

In the UNITY model, defined by Chandy and Misra [3], a program consists of a set of *statements* that access a global shared memory. At each step in the execution, a statement is selected and executed, possibly updating the memory. Superposition in UNITY is defined to be a program transformation that adds a layer on top of a program, while preserving all the properties of the underlying program. Essentially, the transformation modifies the underlying program by augmenting it with a set of new variables and additional statements. In order to preserve the properties of the underlying program, the additional statements must not write to the original variables (although

they may read them). Unfortunately, encapsulation is lacking in UNITY because the interfaces between program modules are not described in terms of well-defined sets of actions, but only in terms of the program variables that they access. A *union* operator is provided for combining modules (by taking the union of the statements and the variables over all modules to be combined). However, one cannot reason about resulting program in terms of actions that occur at module boundaries, but must reason in terms of the memory locations that modules read and write. That is, one cannot treat a module as an abstraction with a certain set of behaviors, but must always be concerned with the internal state of the module. Furthermore, unlike the I/O automaton model, UNITY has no notion of an action being an output of one component and an input to another. We would like such a separation for describing distributed systems.

Thus, UNITY has a superposition mechanism but no encapsulation, while I/O automata provides composition and encapsulation but no superposition mechanism. Therefore, in this paper, we extend the I/O automaton model to permit superposition of program modules. Rather than viewing superposition as a program transformation, we view it as a particular method for combining separate program modules. When one module is superposed on another, the higher level module is allowed to observe (but not modify) the state of the underlying module, while the state of the higher level is unknown to the underlying module. We define an operator for superposing one I/O automaton on another, and show that superposition does not affect the set of executions of the underlying module, thus preserving all properties of that module. A formal specification mechanism is presented that allows the set of correct behaviors of the higher level module to be expressed in terms of the state of the underlying module.

Recently, there has been growing interest in unified models that support superposition. For example, Bougé and Francez [1] argue in favor of the use of superposition as a language construct by describing a number of important applications for its use, and they offer a compositional approach to superposition with a syntactic representation in CSP. In [6], Francez and Forman present a formal semantics for superposition that, like ours, treats superposition as a way of combining programs, rather than a transformation of programs. In [5], the combination of composition and superposition is used for program verification in UNITY [3]. Since UNITY does not provide encapsulation, the verification method presented in [5] lacks compositionality properties, but it is modular in the sense that it is based on separate verification of each component.

A different approach to adding superposition to the I/O automaton model is presented by Nour [16]. In that work, a restricted class of I/O automata, called UNITY automata, is defined in order to express UNITY programs as I/O automata. A superposition operator is defined for this restricted class. Since UNITY automata are restricted to have output actions only, it is not possible to model a superposition in which the higher level module may share actions with the lower level module. In the present work, we do not need such restrictions. In fact, our example algorithm makes important use of shared actions between layers.

The I/O automaton model has been extended previously by Goldman and Lynch to permit automata to make atomic accesses to shared variables [7]. The variables are modeled as being completely external to the automata sharing them, so an automaton must be prepared to observe any value in the memory whenever it executes an access. In the model presented here, variables are also shared, but the sharing relationship is different from that described in [7]. The higher level module sees the variables of the lower level module *at all times*. It is not necessary for the higher level automaton to execute a particular action in order to observe the values of those variables. Therefore, the set of actions “enabled” in the higher level module may change as the lower level module updates its variables. This sort of relationship cannot be modeled using the atomic shared memory extensions of Goldman and Lynch. However, since the two extensions are fully compatible, we have a unified model in which it is possible to construct systems of I/O

automata that use both superposition and shared memory.

The remainder of this paper is organized as follows. In Section 2, we provide a brief introduction to the I/O automaton model. Then, in Section 3, we introduce extensions to that model that enable it to be used for describing problems and algorithms for which layering is the most natural structure. In Section 4, we illustrate the ideas by using the extended model to describe and prove correct the global snapshot algorithm of Chandy and Lamport [2]. In this example, composition, superposition, and encapsulation each play an important role.

2 The I/O Automaton Model

The I/O Automaton model [14] encourages one to write precise statements of the problems to be solved by modules in concurrent systems, allows very careful algorithm descriptions, and can be used to construct rigorous correctness proofs. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results. The I/O automaton model is significantly different from CCS [15] and CSP [9] in that input and output actions in the I/O automaton model are distinguished, and an I/O automaton cannot block an input action from occurring. In that sense, I/O automata are similar to I/O-systems [10, 11, 12]. The following introduction to the model is adapted from [14], which explains the model in more detail, presents examples, and includes comparisons to other models. Readers already familiar with the I/O automaton model may skip this section without loss of continuity.

2.1 I/O Automata

I/O automata are best suited for modeling 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 $s_0, \pi_1, s_1, \dots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \dots$ 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 $s_0 \in \text{start}(A)$. 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 $\text{execs}(A)$, $\text{finexecs}(A)$, $\text{scheds}(A)$, $\text{finscheds}(A)$, $\text{behs}(A)$, and $\text{finbehs}(A)$, 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*.

2.2 Composition

We can construct an automaton modeling a complex system by composing automata modeling 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. $\text{out}(S_i) \cap \text{out}(S_j) = \emptyset$,
2. $\text{int}(S_i) \cap \text{acts}(S_j) = \emptyset$, and
3. no action is contained in infinitely many sets $\text{acts}(S_i)$.

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

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

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:¹

- $\text{sig}(A) = \prod_{i \in I} \text{sig}(A_i)$,
- $\text{states}(A) = \prod_{i \in I} \text{states}(A_i)$,
- $\text{start}(A) = \prod_{i \in I} \text{start}(A_i)$,

¹Here $\text{start}(A)$ and $\text{states}(A)$ are defined in terms of the ordinary Cartesian product, while $\text{sig}(A)$ is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the i th component of the state vector \vec{s} .

- $steps(A)$ is the set of triples $(\bar{s}_1, \pi, \bar{s}_2)$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\bar{s}_1[i], \pi, \bar{s}_2[i]) \in steps(A_i)$, and if $\pi \notin acts(A_i)$ then $\bar{s}_1[i] = \bar{s}_2[i]$, and
- $part(A) = \cup_{i \in I} part(A_i)$.

Given an execution $\alpha = \bar{s}_0 \pi_1 \bar{s}_1 \dots$ of A , let $\alpha|A_i$ (read “ α projected on A_i ”) be the sequence obtained by deleting $\pi_j \bar{s}_j$ when $\pi_j \notin acts(A_i)$ and replacing the remaining \bar{s}_j by $\bar{s}_j[i]$.

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.

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 $fairscheds(A)$.

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 $\beta|\Phi \in \mathcal{P}$, $\pi \in \text{out}(M)$, and $\beta\pi|M \in \text{finbehs}(M)$. 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 Superposition Extensions

In this section, we present definitions that extend the I/O automaton model for superposition of program modules. We begin by defining what it means for an automaton to be “unconstrained” for a particular set of variables, and use this definition to state the requirements for one automaton to be “superposable” on another. We then define the superposition operator, and show that the superposition of one I/O automaton on another produces a new I/O automaton. Therefore, all the standard definitions and results for I/O automata (for fairness, compositionality, etc.) immediately carry over to superposed automata. Furthermore, we show that any fair execution of a superposed automaton, when projected on the underlying module, is a fair execution of the underlying module. In addition, if no output actions of the higher level module are input actions of the underlying module, then every execution of the underlying module is a projection of some execution of the superposed automaton. These results correspond to the UNITY notion that superposition preserves all properties of the underlying algorithm. In addition, we show that when an automaton A is superposed on some other automaton, then the set of schedules of the resulting automaton when projected on the signature of A is a subset of the schedules of A alone. Finally, we present a new problem specification mechanism that is analogous to schedule modules for ordinary I/O automata, but that allows one to specify the allowable behaviors of a higher level module in terms of the *state* of the lower level module. An example illustrating these extensions is presented in Section 4.

Throughout this paper, we refer to the state of an automaton as being divided into sets of variables, where each set of variables takes on values from a particular domain. For example, we may say that the state of automaton A is divided into two sets of variables X and Y with domains $\text{dom}(X)$ and $\text{dom}(Y)$, respectively. In this case, we use an ordered pair (x, y) to name a particular state of A , where $x \in \text{dom}(X)$ and $y \in \text{dom}(Y)$, and we take the set of possible states of A to be the cartesian product $\text{dom}(X) \times \text{dom}(Y)$. If s is a particular state of A , we let $s|X$ denote the values of the variables of X in state s .

All extensions defined in the section are simply additions to the I/O automaton model. We do not redefine any concepts of the original model, so all of its properties carry over to the extended model.

3.1 Unconstrained Automata

When we superpose one module on another, we would like the higher level module not to interfere with the lower level one. In particular, we do not want the higher level module to place constraints on how the lower level module may modify its own variables. Therefore, we will define superposition to apply only when the higher level module is “unconstrained” for the variables of the lower level module. We first define formally what it means for an automaton to be unconstrained for a set of variables. Let X be a set of variables with domain $\text{dom}(X)$. An *unconstrained automaton* A for X is an I/O automaton such that there exists a set P of variables with a set of possible initial values $\text{init}(P)$ such that:

- $states(A) = dom(P) \times dom(X)$,
- $start(A) = init(P) \times dom(X)$, and
- for every step $((p', x'), \pi, (p, x))$ in $steps(A)$, for all $\hat{x} \in dom(X)$, $((p', x'), \pi, (p, \hat{x}))$ is in $steps(A)$.

One may think of P as the set of private variables of the higher level automaton, and of X as the variables of the underlying automaton. Informally, the extra condition on the transition relation says that automaton A places no restrictions on the values of the variables in X following any action. Note, however, that the set of locally-controlled actions enabled in a given state of A may depend on the values of the X variables in that state.

Since an unconstrained automaton is an I/O automaton, all the standard I/O automaton definitions for executions, schedules, behaviors, and composition carry over to unconstrained automata. One may think of an “ordinary” I/O automaton as an unconstrained automaton for $X = \emptyset$.

One way to model a layered multicomponent system is to individually superpose pairs of automata and then compose. An equally valid method is to create two entire system layers through composition, and then superpose. In using the latter method, we would like the composition of an unconstrained automaton for X and an unconstrained automaton for Y , with $X \cap Y = \emptyset$, to be an unconstrained automaton for $X \cup Y$. However, this is not the case. Even if the components of the higher layer are each appropriately unconstrained, their composition is not.² Therefore, we define a relaxation operator \mathcal{U} that builds an unconstrained automaton from an ordinary one. Let A be an I/O automaton whose state is divided into two sets of variables P and X with domains $dom(P)$ and $dom(X)$ respectively. We define the *relaxation of A with respect to X* , denoted $\mathcal{U}(A, X)$, to be the automaton B as follows:

- $sig(B) = sig(A)$,
- $states(B) = states(A)$,
- $start(B) = \{(p, \hat{x}) : \hat{x} \in dom(X) \wedge \exists x \in dom(X), (p, x) \in start(A)\}$,
- $steps(B) = \{((p', x'), \pi, (p, \hat{x})) : \hat{x} \in dom(X) \wedge \exists x \in dom(X), ((p', x'), \pi, (p, x)) \in steps(A)\}$,
and
- $part(B) = part(A)$.

The relaxation operator \mathcal{U} simply constructs the new automaton by adding enough start states and steps to make it unconstrained for X . The following lemma follows immediately from the definitions.

Lemma 1: Let A be an I/O automaton whose state is divided into two sets of variables, P and X . Then $\mathcal{U}(A, X)$ is an unconstrained automaton for X .

The following result allows us to prove properties of the schedules of individual unconstrained automata with the knowledge that these properties will carry over to all schedules of the relaxation of the composition.

²For example, suppose A_X is an unconstrained automaton for X and A_Y is an unconstrained automaton for Y . In the composition of A_X and A_Y , the values of the variables of X are changed only in steps involving actions of A_X . Therefore, any action of A_Y that is not an action of A_X is constrained to leave the values of the variables in X unchanged. Thus, the composition of A_X and A_Y is not unconstrained for $X \cup Y$.

Lemma 2: Let $\{X_i\}_{i \in \mathcal{I}}$ be a set of disjoint sets of variables, and let $\{A_i\}_{i \in \mathcal{I}}$ be a collection of strongly compatible automata, where each A_i is unconstrained for X_i . Let A be the composition $\Pi_{i \in \mathcal{I}} A_i$, and let A_u be the automaton $\mathcal{U}(A, \bigcup_{i \in \mathcal{I}} X_i)$. Then $\text{scheds}(A_u) = \text{scheds}(A)$ and $\text{fairscheds}(A_u) = \text{fairscheds}(A)$.

Proof: We know that $\text{scheds}(A) \subseteq \text{scheds}(A_u)$, since $\text{start}(A) \subseteq \text{start}(A_u)$ and $\text{steps}(A) \subseteq \text{steps}(A_u)$ by definition. We show that $\text{scheds}(A_u) \subseteq \text{scheds}(A)$ using the following construction. Let α_u be an execution of A_u . For each state s_u of α_u , and for each $i \in \mathcal{I}$, let $\text{next-step}_i(s_u)$ be the state of α_u immediately preceding the first action of A_i that occurs in α_u after state s_u . (If no action of A_i follows s_u , then $\text{next-step}_i(s_u)$ is the state immediately after the last action of A_i in α_u . If no action of A_i occurs in α_u , then $\text{next-step}_i(s_u)$ is the initial state of α_u .) We construct α to be identical to α_u , except that $\forall i \in \mathcal{I}, \forall n > 0$, if s_u is the n^{th} state of α_u and s is the n^{th} state of α , then $s|X_i = \text{next-step}_i(s_u)|X_i$. Note that the value of $s|X_i$ is identical for all states s between successive actions of A_i in α , and is equal to the value of X_i just before the next step of A_i in α_u .

Clearly $\text{sched}(\alpha) = \text{sched}(\alpha_u)$. To show that α is a schedule of A , we must show that (1) if s_0 is the first state of α , then $s_0 \in \text{start}(A)$, and (2) every step (s', π, s) in α is in $\text{steps}(A)$. For condition (1), since each component A_i is unconstrained for X_i , we know that the initial value for x_i may be any value in $\text{dom}(X_i)$. Therefore, $s_0 \in \text{start}(A)$. For condition (2), we note that if (s', π, s) is the n^{th} step of α , we know from the construction that if $\pi \in \text{acts}(A_i)$, then $s'|A_i = s'_u|A_i$, where s'_u is the n^{th} state of α_u , and that π is enabled in state s'_u . Therefore, π is enabled in state s' . And since A_i is unconstrained for X_i , any value is possible for X_i in the resulting state. Furthermore, we know from the construction that if $\pi \notin \text{acts}(A_i)$, then $s|A_i = s'|A_i$. Therefore, (s', π, s) is a step of A .

The fairness result follows from the above arguments and the fact that $\text{part}(A_u) = \text{part}(A)$ by definition of the relaxation operator. \blacksquare

3.2 Superposition

In this section, we define the conditions under which one module may be superposed on another, and then define the superposition operator itself.

Requirements for Superposition: In order to provide a sensible semantics for the superposition operator, we define the superposition of one automaton on another only when the two automata satisfy certain compatibility conditions, defined as follows. Let X be a set of variables with domain $\text{dom}(X)$. We say that automaton A is *superposable* on automaton B with respect to X iff

1. A is unconstrained for X ,
2. $\text{states}(B) = \text{dom}(X)$, and
3. $\text{sig}(A)$ and $\text{sig}(B)$ are strongly compatible.

Loosely speaking, the first condition ensures that module B may freely modify its own variables in the superposition. The second condition says that the set of states of the underlying automaton must match the domain for the set of variables on which A is unconstrained. The third condition is the usual restriction for composition of automata.

Superposition Operator: We would like superposition to capture the idea that the higher level automaton is allowed to observe (but not modify) the state of the lower level automaton, and that

the lower level automaton is unaware of the variables of the higher level automaton. We want the actions of the superposed automaton to include the actions of both the high level and low level automata, and we wish to allow the possibility of actions that are shared by both automata. This motivates the following definition.

Let X be a set of variables with domain $dom(X)$, and let A and B be automata such that A is superposable on B with respect to X . We define the *superposition of A on B with respect to X* , denoted $C = S(A, B, X)$, as follows:

- $sig(C) = sig(A) \times sig(B)$ (usual signature composition),
- $states(C) = states(A)$,
- $start(C) = \{(p, x) \in start(A) : x \in start(B)\}$,
- $steps(C) =$ all steps $((p', x'), \pi, (p, x))$ such that the following conditions hold:
 1. $\pi \in sig(C)$
 2. if $\pi \in sig(A)$, then $((p', x'), \pi, (p, x)) \in steps(A)$
 3. if $\pi \in sig(B)$, then $(x', \pi, x) \in steps(B)$
 4. if $\pi \notin sig(A)$, then $p = p'$
 5. if $\pi \notin sig(B)$, then $x = x'$, and
- $part(C) = part(A) \cup part(B)$.

Informally, the signature of the superposed automaton C is the composition of the signatures of A and B . The states of C are the same as the states of A , and the set of start states of C is the set of all start states of A such that the values of X agree with some start state of B . The most interesting part of the superposition definition is the construction of the set of steps. It says that any step of C for an action of A must also be a step of A . Similarly, any step of C for an action of B must be a step of B , when projected on the variables in X . Essentially, the actions of A and B are enabled just as before, with automaton B controlling the values of the variables in X . The last two conditions of the $steps(C)$ construction simply prevent steps involving only B from modifying the private variables of A , and steps involving only A from modifying the variables in X . That is, if a step of C does *not* involve an action of A , then the private state variables of A must not be modified by the step. Similarly, if a step of C does *not* involve an action of B , then the values of the variables in X are unchanged by the step.

In a step for an action shared by A and B , the private state of A is modified according to the transition relation of A , while the state of X is modified according to the transition relation of B . This should agree with one's intuition about the semantics for such shared actions.

The following lemma states that a superposition of one I/O automaton on another results in a new I/O automaton. This implies that all the standard definitions and results for I/O automata, notably for composition and fairness, immediately carry over to superposed automata.

Lemma 3: Let X be a set of variables with domain $dom(X)$, and let A and B be automata such that A is superposable on B with respect to X . Then $C = S(A, B, X)$ is an I/O automaton.

Proof: We must show that inputs of C are always enabled. That is, we must show that for all states $s' \in states(C)$ and for all actions $\pi \in in(C)$, there exists a state $s \in states(C)$ such that $(s, \pi, s) \in steps(C)$. Let $s' = (p', x')$. There are three cases for $\pi \in in(C)$. For each case, we exhibit an appropriate new state s :

1. $\pi \in \text{sig}(A)$ and $\pi \notin \text{sig}(B)$. Since A is an unconstrained automaton, we know that $\exists p \in \text{private}(A)$ such that $\forall \hat{x} \in \text{dom}(X)$, $((p', x'), \pi, (p, \hat{x})) \in \text{steps}(A)$. Specifically, if we let $\hat{x} = x'$, then we are done.
2. $\pi \notin \text{sig}(A)$ and $\pi \in \text{sig}(B)$. Since B is an I/O automaton, we know that $\exists x \in \text{states}(B)$ such that $(x', \pi, x) \in \text{steps}(B)$. Therefore, since $\pi \notin \text{sig}(A)$, $((p', x'), \pi, (p', x)) \in \text{steps}(C)$.
3. $\pi \in \text{sig}(A)$ and $\pi \in \text{sig}(B)$. Since A is an unconstrained automaton, we know that $\exists p \in \text{private}(A)$ such that $\forall \hat{x} \in \text{dom}(X)$, $((p', x'), \pi, (p, \hat{x})) \in \text{steps}(A)$. Furthermore, since B is an I/O automaton, we know that $\exists x \in \text{states}(B)$ such that $(x', \pi, x) \in \text{steps}(B)$. Therefore, letting $\hat{x} = x$ completes the proof.

In each case, π is enabled from s' . ■

The following two results formalize the notion that properties of the underlying algorithm are preserved in the superposition.

Lemma 4: Let X be a set of variables with domain $\text{dom}(X)$. Let A and B be automata such that A is superposable on B with respect to X , and let $C = \mathcal{S}(A, B, X)$. Then $\text{execs}(C)|B \subseteq \text{execs}(B)$ and $\text{fairexecs}(C)|B \subseteq \text{fairexecs}(B)$.

Proof: Let α be a (fair) execution of C . By definition of superposition, if (s', π, s) is a step of α and $\pi \notin \text{acts}(B)$ then $s|X = s'|X$. Therefore, $\alpha|B$ is a (fair) execution of B . (The fairness result follows from the fact that $\text{part}(B) \subseteq \text{part}(C)$, so any execution fair to the classes of C must also be fair to the classes of B .) ■

In general, it is not the case that every execution of the lower level automaton is a projection of an execution of the superposed automaton. For example, lower level automaton B may have π as an input action, so its set of executions include executions in which π occurs multiple times. If automaton A is defined to have π as an output action such that π occurs at most once in every execution of A , then none of the executions of B in which π occurs more than once are projections of executions of the superposition of A on B . However, when no output actions of the higher level automaton are inputs to the lower level automaton, the converse of Lemma 4 holds, and we have the following result.

Lemma 5: Let X be a set of variables with domain $\text{dom}(X)$. Let A and B be automata such that A is superposable on B with respect to X , and let $C = \mathcal{S}(A, B, X)$. If $\text{in}(B) \cap \text{out}(A) = \emptyset$, then $\text{execs}(C)|B = \text{execs}(B)$ and $\text{fairexecs}(C)|B = \text{fairexecs}(B)$.

Proof: Lemma 4 tells us that $\text{execs}(C)|B \subseteq \text{execs}(B)$ and $\text{fairexecs}(C)|B \subseteq \text{fairexecs}(B)$. Let β be a (fair) execution of B . Since $\text{in}(B) \cap \text{out}(A) = \emptyset$, we know that the higher level component A has no control over which actions of $\text{acts}(B)$ occur in an execution of C . Furthermore, only the actions of B may change the variables in X in the superposition. Therefore, since a locally-controlled action of B is enabled from state s in C iff it is enabled from state $s|X$ in B , there must exist some (fair) execution γ of C such that $\gamma|B = \beta$. Thus, $\text{execs}(B) \subseteq \text{execs}(C)|B$ and $\text{fairexecs}(B) \subseteq \text{fairexecs}(C)|B$. ■

The next result says that when an automaton A is superposed on some other automaton, then the set of schedules of the resulting automaton, when projected on the signature of A , is a subset of the schedules of A alone. This is very important because it allows us to prove safety properties about A alone with the knowledge that these properties will hold when A is superposed on some other automaton.

Lemma 6: Let X be a set of variables with domain $dom(X)$. Let A and B be automata such that A is superposable on B with respect to X , and let $C = S(A, B, X)$. Then $scheds(C)|sig(A) \subseteq scheds(A)$.

Proof: Let γ be an execution of C . We construct α from γ by the following steps:

1. Remove from γ all actions not in $sig(A)$. This may create sequences of states not separated by actions.
2. Replace the sequence of states between each pair of successive actions by the last state in that sequence.

Clearly, α is an alternating sequence of states of A and actions of A . To show that $\alpha \in execs(A)$, we must show that the first state of α is in $start(A)$ and that if $\sigma = ((p', x'), \pi, (p, x))$ is a step in α , then $\sigma \in steps(A)$. For both of these, we use the following fact.

Fact: If a sequence of states from γ is replaced in step 2 of the construction by the single state (p, x) , then every state in that sequence has p as the value of the private state of A .

Proof of Fact: In γ , each of these states is separated by an action not in the signature of A . From the definition of superposition, we know that any step in γ not involving an action of A does not change the values of the private variables of A . Since (p, x) is the last state in the sequence, every state in the sequence must have p as its first component.

From the above fact, we know that the first state of γ and the first state of α agree on the values of the private variables of A . Since A is unconstrained, any value in $dom(X)$ is an allowable value for the second component of the start state. Therefore, the first state of α is a start state of A .

If $\sigma = ((p', x'), \pi, (p, x))$ is a step in γ , then we know that π occurs from state (p', x') in α . Furthermore, from the above fact, we know that π results in state (p, \hat{x}) in γ for some $\hat{x} \in dom(X)$. So, we know that $steps(A)$ contains the step $((p', x'), \pi, (p, \hat{x}))$ for some $\hat{x} \in dom(X)$. Therefore, since A is unconstrained for X , we know that $steps(A)$ contains the step $((p', x'), \pi, (p, \hat{x}))$ for all $\hat{x} \in dom(X)$, and specifically for $\hat{x} = x$. This completes the proof. ■

Note that not all schedules of A are necessarily possible in the superposition, since certain states reachable in A alone may not be reachable in the superposition. For example, suppose a particular action π of A is enabled only when a variable $x \in X$ has a particular value v , and suppose that the automaton B on which A is superposed is defined to never set $x = v$. Since A alone may set x to any value (by the definition of unconstrained for X), the action π may occur in behaviors of A . However, by the definition of superposition, there is no step of $S(A, B, X)$ that results in $x = v$, so π is never enabled. This is a perfectly natural and desirable property of superposition, for it says that the state of the lower layer affects the behavior of the higher layer.

Another interesting fact is that $fairscheds(C)|sig(A)$ and $fairscheds(A)$ are incomparable. We know from the above paragraph that $fairscheds(A) \not\subseteq fairscheds(C)|sig(A)$. But it is also the case that $fairscheds(C)|sig(A) \not\subseteq fairscheds(A)$, as witnessed by the following example. Suppose that A has only two actions, π_1 and π_2 , each in its own class of the partition. Furthermore, suppose that both events are enabled exactly when $x = 0$. Now, suppose that B has exactly one action π_3 , that toggles the value of x between 0 and 1, and is always enabled. In the superposition of A on B , a fair schedule would be $\pi_1, \pi_3, \pi_3, \pi_1, \pi_3, \pi_3, \pi_1, \dots$, in which the class containing π_2 is given a chance to take a step only when $x = 1$. However, the infinite schedule $\pi_1, \pi_1, \pi_1, \dots$ is *not* a fair schedule

of A alone: Since the schedule consists of infinitely many π_1 actions, it must be that π_1 is enabled from every state of the corresponding execution. Therefore, $x = 0$ in all states of that execution. But π_2 is also enabled whenever $x = 0$, yet the class containing π_2 is never given a chance to take a step, so the schedule is not fair.

The reason for the above fact is that the preconditions for the locally-controlled actions of A are allowed to depend upon the values of the variables in X . Keeping this in mind, consider the following additional condition on unconstrained automata. If A is an unconstrained automaton for X , then A is said to be *completely unconstrained for X* iff for all actions $\pi \in \text{sig}(A)$, if $((p', x'), \pi, (p, x)) \in \text{steps}(A)$ then for all $\hat{x} \in \text{dom}(X)$, there exists a state $(\hat{p}, \hat{x}) \in \text{states}(A)$ such that $((p', \hat{x}), \pi, (\hat{p}, \hat{x})) \in \text{steps}(A)$. In other words, whether or not an action of A is enabled can depend only upon the values of its private variables. The only way for A to make any use of the variables of X would be for it to modify its own local variables according to what it observes in X , causing other actions of A to become enabled or disabled. Modifying the definition of superposable to require the higher level automaton to be completely unconstrained for X would allow us to prove that $\text{fairscheds}(C) \upharpoonright \text{sig}(A) \subseteq \text{fairscheds}(A)$, but would result in a significant loss of expressive power. So, rather than require this condition outright, we state the following lemma, which says that if an automaton happens to be completely unconstrained, then the containment result holds for its fair schedules. This gives us more flexibility in the use of the model.

Lemma 7: Let X be a set of variables with domain $\text{dom}(X)$. Let A and B be automata such that A is completely unconstrained for X and A is superposable on B with respect to X . Let $C = S(A, B, X)$. Then $\text{fairscheds}(C) \upharpoonright \text{sig}(A) \subseteq \text{fairscheds}(A)$.

Proof: Analogous to that of Lemma 6, but noting that the actions of A are enabled independently of the value of X and applying the definition of fairness. ■

The definition of an unconstrained automaton A for X requires that the value of X may be changed arbitrarily with each step of A . However, a more natural way to describe the behaviors of a module to be superposed on another module might be to allow the values of X to change *between* the steps of A as well. For this, we define the notion of an “extended execution” in which several states may occur between two successive actions. If A is an unconstrained automaton for X , we define an *extended execution* of A to be a sequence α of states in $\text{states}(A)$ and actions in $\text{acts}(A)$, beginning with a state in $\text{start}(A)$, such that:

1. if a state-action-state sequence $s'\pi s$ appears in α , then (s', π, s) is in $\text{steps}(A)$,
2. if two states s' and s appear consecutively in α , then they differ only in the value of X , and
3. no two actions appear consecutively in α .

We define fairness for extended executions exactly as for ordinary executions. We let $\text{extexecs}(A)$ and $\text{fairextexecs}(A)$ denote the sets of extended executions and fair extended executions of A , respectively. If α is a sequence of states and actions and Π is a set of actions, we define the notation $\alpha \upharpoonright \Pi$ to be the sequence that results from deleting from α exactly those actions not in Π . Using extended executions instead of ordinary executions, we get the desired fairness result:

Lemma 8: Let X be a set of variables with domain $\text{dom}(X)$. Let A and B be automata such that A is superposable on B with respect to X , and let $C = S(A, B, X)$. Then $\text{execs}(C) \upharpoonright \text{sig}(A) \subseteq \text{extexecs}(A)$ and $\text{fairexecs}(C) \upharpoonright \text{sig}(A) \subseteq \text{fairextexecs}(A)$.

Proof: Let α be an execution of C . From the definition of superposition, we know that α begins with a state in $\text{start}(A)$, and that any step (s', π, s) occurring in α with $\pi \in \text{sig}(A)$ must be a step

of A . Also by the definition of superposition, for any step (s', π, s) where π is *not* in $\text{sig}(A)$, s' and s must differ only in the value of X . Therefore, $\alpha \parallel \text{sig}(A)$ is an extended execution of A by definition. If α is fair, then since α and $\alpha \parallel \text{sig}(A)$ contain the same sequence of states, and since $\text{part}(A) \subseteq \text{part}(C)$, we know that $\alpha \parallel \text{sig}(A)$ is a fair extended execution of A . ■

3.3 Partial Execution Modules

It is important to have a formal mechanism for specifying the problem to be solved by an automaton. Schedule modules, as described in Section 2.4, permit us to specify the allowable schedules of a module in terms of the actions that occur the boundary with its environment. However, if an automaton A is to be superposed on top of some underlying automaton B , then we would like to specify the allowable behaviors of A not only in terms of the actions that occur at its external interface, but also in terms of the internal state of B . To accomplish this, we define a new specification mechanism called a “partial execution module.”

Let X be a set of variables with domain $\text{dom}(X)$, and let Π be a set of actions. A *partial execution* for Π and X is defined to be a sequence of states and actions, beginning with a state, such that each state is in $\text{dom}(X)$, each action is in Π , and each action is immediately followed by a state. Note that a partial execution may contain several states between two consecutive actions.

A *partial execution module* H consists of

- $\text{sig}(H)$, an external action signature,
- $\text{vars}(H)$, a set of variables with domain $\text{dom}(\text{vars}(H))$, and
- $\text{pexecs}(H)$, a set of partial executions for $\text{sig}(H)$ and $\text{vars}(H)$.

A partial execution module H defines a problem to be solved by an unconstrained automaton for $\text{vars}(H)$ with external signature $\text{sig}(H)$. In order to define what it means for an automaton to “solve” H , we need a way to extract partial executions from extended executions. Let X be a set of variables with domain $\text{dom}(X)$, let Π be a set of actions, and let α be an extended execution of any automaton that is unconstrained for X . We define $\alpha \parallel (\Pi, X)$, the *partial execution for Π and X in α* , to be the same as α , except that each state s is replaced by its projection on X and each action not in Π is deleted. If A is an unconstrained automaton for X with external signature Π , we define $\text{pexecs}(A, X)$ to be the set $\{\alpha \parallel (\Pi, X) : \alpha \in \text{fairextexecs}(A)\}$.

An automaton A is said to *solve* a partial execution module H iff $\text{pexecs}(A, \text{vars}(H)) \subseteq \text{pexecs}(H)$.

3.4 Superposition for Partial Executions

Lynch and Tuttle define composition for both automata and schedule modules. So far, we have defined the superposition of one automaton on another, but have not yet defined an analogous operator for superposing a partial execution module on another module. We now complete the theory by defining the superposition of a set of partial executions on a set of ordinary executions.

Let X be a set of variables and let Π and Δ be sets of actions. Let ∂ be a set of partial executions for Π and X , and let Φ be a set of alternating sequences of states in $\text{dom}(X)$ and actions in Δ . Let \mathcal{U} be the set of all alternating sequences of states of X and actions of $\Pi \cup \Delta$. We now define the *superposition of ∂ on Φ with respect to X* . Overloading the \mathcal{S} notation, we define

$$\mathcal{S}(\partial, \Phi, X) = \{\alpha \in \mathcal{U} : \alpha \parallel \Pi \in \partial \wedge \alpha \parallel \Delta \in \Phi\}.$$

In other words, for each element α of $\mathcal{S}(\partial, \Phi, X)$, deleting all actions from α except those in Π results in a partial execution in ∂ , and projecting α on the actions of Δ results in an execution in Φ .

The following result says that the set of fair behaviors of a superposition of A on B with respect to X is the same as the set of behaviors resulting from the superposition of $\text{pexecs}(A, X)$ on the fair executions of B .

Lemma 9: Let X be a set of variables. If automaton A is superposable on automaton B with respect to X , then $\text{fairbehs}(\mathcal{S}(A, B, X) | (\Pi, X)) = \text{behs}(\mathcal{S}(\text{pexecs}(A), \text{fairexecs}(B), X))$.

Proof: If β is a fair behavior of $\mathcal{S}(A, B, X)$, let α be the corresponding execution. By Lemma 8, $\alpha | \text{sig}(A)$ is a fair extended execution of A , so $\alpha | (\text{sig}(A), X) \in \text{pexecs}(A)$. And by Lemma 4, $\alpha | B \in \text{fairexecs}(B)$, so $\text{fairbehs}(\mathcal{S}(A, B, X) | (\Pi, X)) \subseteq \text{behs}(\mathcal{S}(\text{pexecs}(A), \text{fairexecs}(B), X))$.

To show the other direction, let α be an element of $\mathcal{S}(\text{pexecs}(A), \text{fairexecs}(B), X)$. From the definition of superposition of partial executions, we know that $\alpha | \Pi \in \text{pexecs}(A)$. Therefore, there exists a fair extended execution α' of A such that $\alpha = \alpha' | (\text{ext}(A), X)$. Also from the definition of superposition of partial executions, we know that $\alpha | \Delta \in \text{fairexecs}(B)$. Since the states of α and α' are identical with respect to X , we know that for each step (s', π, s) of $\alpha | \Delta$, there exists in α' a pair of consecutive states \hat{s}' and \hat{s} such that $\hat{s}' | X = s'$ and $\hat{s} | X = s$. We construct α'' by inserting each action of $\alpha | \Delta$ between the corresponding pair of states in α' . To complete the proof, we must show that α'' is a fair execution of $\mathcal{S}(A, B, X)$. We know that α'' begins with an initial state of A . Now, we consider the four possible cases for each step (s', π, s) in α'' :

1. If $\pi \in \text{sig}(A)$, then $(s', \pi, s) \in \text{steps}(A)$, since α' is an extended execution of A .
2. If $\pi \in \text{sig}(B)$, then $(s' | X, \pi, s | X) \in \text{steps}(B)$, because of our construction of α'' from α and the fact that $\alpha | \Delta$ is an execution of B .
3. If $\pi \notin \text{sig}(A)$, then s' and s differ only in the value of X , by definition of an extended execution.
4. If $\pi \notin \text{sig}(B)$, then $s' | X = s | X$, again because $\alpha | \Delta$ is an execution of B .

Therefore, α'' is an execution of $\mathcal{S}(A, B, X)$. To show that α'' is fair, we note that $\text{part}(\mathcal{S}(A, B, X)) = \text{part}(A) \cup \text{part}(B)$ by the definition of superposition, and we consider the classes of A and B separately. We know that α' is a fair extended execution of A . Therefore, since $\alpha'' | A = \alpha'$, α'' is fair to the classes of A . Similarly, since $\alpha | \Delta$ is a fair execution of B , and $\alpha'' | B = \alpha | \Delta$, we know that α is fair to the classes of B . ■

4 Example: Global Snapshot

Extended with the superposition operator, the I/O automaton model can now be used to model algorithms for problems in which layering is the most natural structure. In this section, we use the extended model to describe and prove correct the global snapshot algorithm of Chandy and Lamport [2]. We begin by defining the global snapshot problem as a partial execution module G . Then we describe the global snapshot algorithm as an automaton to be superposed on an arbitrary application program. Finally, we give a complete proof that the global snapshot algorithm solves partial execution module G .

Both composition and superposition are important in this example because the snapshot automaton is, in fact, the composition of a collection of automata, one superposed on each component

of the distributed application. Each of these automata acts as a “filter” of the incoming and outgoing messages of the application program. Since superposition permits each of these automata to observe the state of the corresponding component of the application, the existence of the global snapshot algorithm is entirely transparent to the application program. Without the superposition operator, it would have been necessary to modify the application program to explicitly communicate its state to the snapshot algorithm at the appropriate point in the global snapshot protocol. The correctness proof shows that the global snapshot automaton solves the partial execution module and makes important use of the compositionality results of the model.

4.1 Problem Specification

We consider systems of processes that communicate by sending messages over a network. The network guarantees eventual one-time delivery of each message such that messages sent from a given process to each other process arrive in the order sent (pairwise FIFO). The goal of a *distributed global snapshot* protocol is to produce a global state of a system (states of all processes and the set of messages in transit) during an ongoing computation. The snapshot algorithm is not allowed to interfere with the computation of the rest of the system. For example, the snapshot algorithm cannot halt the system. In addition, the snapshot obtained must be both *consistent* and *recent*. By this we mean that if the snapshot protocol is initiated in a system state *initiation*, terminates in the system state *termination*, and produces the snapshot state *snapshot*, then there is some execution of the system containing the states *initiation*, *snapshot* and *termination*, in that order (possibly with other states in between) in which the same set of messages was sent and received by each process. Note that the state *snapshot* may not ever actually occur. The requirement is only that it “could” have occurred between the *initiation* and *termination* states by a suitable reordering of the actions.

The global snapshot problem, and protocols for solving it, are neatly specified using the superposition definitions defined in the previous section. We view the snapshot algorithm as a layer to be superposed on top of the application layer. We begin by specifying the signature for each of the underlying application processes and a schedule module for the network. Then, we present a partial execution module G that formalizes the problem statement given above.

4.1.1 Application Processes

Let \mathcal{I} be a finite set of names for the communicating processes in the application program. Let \mathcal{M} be a universal set of messages that contains a special marker symbol ($\#$) used by the snapshot protocol but never sent as a message by the application. For each $i \in \mathcal{I}$, we fix a corresponding application process u_i . Each process u_i is modeled as an automaton having a set of state variables X_i with domain $states(u_i)$, and the following signature:

Input actions: $Rcv(m, j, i), m \in \mathcal{M} - \{\#\}, j \in \mathcal{I}$
Output actions: $SEND(m, i, j), m \in \mathcal{M} - \{\#\}, j \in \mathcal{I}$

The input actions represent u_i receiving a message m from u_j , and the output actions represent u_i sending a message m to u_j . Associated with u_i are two sets, $out-chans(i)$ and $in-chans(i)$, both subsets of \mathcal{I} . One may think of $out-chans(i)$ as identifying those application processes to which u_i may send messages, the “outgoing channels” of u_i . Similarly, $in-chans(i)$ identifies the application process from which u_i may receive messages, the “incoming channels” of u_i . More precisely, let $CHANS(V, E)$ be a strongly connected graph with $|\mathcal{I}|$ vertices uniquely labeled by the

elements of \mathcal{I} . For each $i \in \mathcal{I}$, we let $out-chans(i) = \{j \in \mathcal{I} \mid (i, j) \in E\}$ and we let $in-chans(i) = \{j \in \mathcal{I} \mid (j, i) \in E\}$.

A sequence β of actions of u_i is said to be *well-formed* for i iff for all $m \in \mathcal{M}$, and for all $j, k \in \mathcal{I}$, if $SEND(m, i, j)$ occurs in β , then $j \in out-chans(i)$ and $m \neq \#$, and if $RCV(m, k, i)$ occurs in β , then $k \in in-chans(i)$.

We require that u_i preserve well-formedness for i , but make no other restrictions on the allowable behaviors of u_i . We make no restrictions on the domain $states(u_i)$.

4.1.2 Correspondence Relations

In specifying the network, as well as the global snapshot problem, we use a correspondence relation technique similar to that of [4]. Let t denote a text string. We define a *message action* for t to be any action of the form $t(m, i, j)$, where $m \in \mathcal{M}$ and $i, j \in \mathcal{I}$. Let β be a sequence of actions, and let x and y be text strings. Let Π_x be the set of events for message actions for x in β , and let Π_y be the set of events for message actions for y in β . Let \mathcal{C} be a binary relation on the set of events in an execution onto itself, and let us say that two events in an execution *correspond* iff the relation is true for that pair of events. We say that \mathcal{C} is a *correspondence relation* for x and y in β iff the first four of the following conditions hold, and is a *live correspondence relation* iff all of the following conditions hold.

1. Corresponding events have identical arguments.
2. Each event $\pi_y \in \Pi_y$ corresponds to exactly one event $\pi_x \in \Pi_x$, and π_x precedes π_y in β .
3. Each event $\pi_x \in \Pi_x$ corresponds to at most one event in Π_y .
4. If $x(m, i, j)$ precedes $x(m', i, j)$ in β , and $y(m, i, j)$ and $y(m', i, j)$ are their corresponding events, then $y(m, i, j)$ precedes $y(m', i, j)$ in β .
5. For each event $\pi_x \in \Pi_x$, there exists a corresponding event in Π_y .

An intuitive explanation of these conditions follows their use in specifying the network. The following lemma states a transitivity property of correspondence relations.

Lemma 10: Let β be a sequence of actions, and let x, y and z be text strings. If \mathcal{C}_{xy} is a (live) correspondence relation for x and y in β , and \mathcal{C}_{yz} is a (live) correspondence relation for y and z in β , then there exists a (live) correspondence relation \mathcal{C}_{xz} for x and z in β .

Proof: Let \mathcal{C}_{xz} be defined as follows. Two events π_x and π_z in β correspond iff there exists an event π_y in β such that π_x corresponds to π_y according to \mathcal{C}_{xy} and π_y corresponds to π_z according to \mathcal{C}_{yz} . The properties of a (live) correspondence relation follow immediately. ■

4.1.3 The Network

Rather than modeling the network as an explicit I/O automaton, we define an action signature for the network and then define a well-formedness property of sequences of those actions that characterizes the desired behaviors of the network. The signature of the network is as follows:

Input actions: $SEND(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$
Output actions: $RCV(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$

If β is a sequence of actions, then β is *network admissible* iff there exists a live correspondence relation for $SEND$ and RCV in β . This means that (1) a $SEND$ and RCV correspond only if

they match on the arguments m , i , and j , (2) each RCV corresponds to exactly one SEND, and the SEND occurs earlier, (3) to each SEND event there corresponds at most one RCV event, (4) messages between pairs of processes are delivered in the order sent, and (5) each message sent is eventually received. The fifth condition is the *liveness* property we assume to be guaranteed by the network.

4.1.4 The Application System

In defining our correctness condition for the global snapshot algorithm, we would like to express the notion that the application processes should not be able to tell whether they are running in a system with the snapshot protocol, or in a system without the snapshot protocol. Therefore, we explicitly define the set of allowable behaviors of the “system without the snapshot protocol” to form the basis of our correctness condition.

Let $U = \Pi_{i \in \mathcal{I}} u_i$, the composition of all the application processes. We define the set of fair behaviors of the application system, denoted $fairbehs(S_{app})$ to be the set of behaviors of all network admissible fair executions of U . We will state the correctness conditions for a global snapshot protocol in terms of $fairbehs(S_{app})$. In doing so, it will be helpful to have the following definitions. Let $seqs(\mathcal{M})$ be the set of all sequences of elements of \mathcal{M} , including the empty sequence ϵ . Let α be element of $fairbehs(S_{app})$, and let \mathcal{C} be a correspondence relation for SEND and RCV in α (we know there is one, by the definition of network admissible). If α' is a prefix of α , we define $in-transit_{\alpha', \mathcal{C}} : (\mathcal{I} \times \mathcal{I}) \rightarrow seqs(\mathcal{M})$ as follows. For each pair $i, j \in \mathcal{I}$, $in-transit_{\alpha', \mathcal{C}}(i, j)$ is the sequence of messages m such that $SEND(m, j, i)$ occurs in α' , but the corresponding $RCV(m, j, i)$ does not, ordered according to the SEND events. In other words, for each i, j pair, we have the sequence of messages sent from u_j to u_i , but not yet delivered to u_i . We define $in-transit$ analogously for the schedule of α , and for executions or partial executions whose schedules, projected on U , are in $fairbehs(S_{app})$.

4.1.5 Partial Execution Module G

In this section, we specify the global snapshot problem by defining a partial execution module G . Let $chans$ name the set of all possible functions from elements of \mathcal{I} to elements of $seqs(\mathcal{M})$, and let $all-chans$ name the set of all possible functions from elements of \mathcal{I} to elements of $chans$. The signature $sig(G)$ is as follows:

Input actions:	$START_i, i \in \mathcal{I}$ $SEND(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$ $MSG_RCV(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$
Output actions:	$MSG_SEND(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$ $RCV(m, i, j), m \in \mathcal{M}, i, j \in \mathcal{I}$ $DONE_i(a, c), i \in \mathcal{I}, a \in states(u_i), c \in chans$

Let α be a sequence of $states(U)$ and actions of G . If a $START$ action (for any i) occurs in α and exactly one $DONE_i$ action occurs in α for each $i \in \mathcal{I}$, then we let $\alpha = \alpha_1 \alpha_2 \alpha_3$, where α_2 begins with the first $START$ action and ends with the last $DONE$ action. Furthermore, we define $snap(\alpha)$ to be the pair $(s \in states(U), c \in all-chans)$ such that $\forall i \in \mathcal{I}$, if $DONE_i(a_i, c_i)$ occurs in α , then $s|u_i = a_i$ and $c(i) = c_i$. In other words, $snap(\alpha)$ is the collective state of the system (including messages in transit) reported in the $DONE_i$ actions that occur in α .

Since the snapshot algorithm has no control over its input actions, we require that G behave properly only when its environment, namely the network and the application processes, is well-

behaved. Let α be a sequence of $states(U)$ and actions of G . We say that α is *admissible* iff (1) there exists a live correspondence relation C_α for MSG_SEND and MSG_RCV in α , and (2) $\forall i \in \mathcal{I}, \alpha|u_i \in execs(u_i)$. We place constraints on the behavior of G only for admissible partial executions.

Let α be a sequence of $states(U)$ and actions of G . Then $\alpha \in pexecs(G)$ iff the following condition holds. If α is admissible, then

1. $sched(\alpha)|U \in fairbehs(S_{app})$, and
2. if a START action occurs in α , then
 - (a) $\forall i \in \mathcal{I}$, exactly one $DONE_i$ occurs in α , and
 - (b) $\exists \beta = \beta_1\beta_2s\beta_3\beta_4$, an execution of U , with correspondence relation C_β for SEND and RCV in β , such that
 - i. $\forall i \in \mathcal{I}, \beta|u_i = \alpha|u_i$,
 - ii. the last state of α_1 is the last state of β_1 , and $in-transit_{\beta_1, C_\beta} = in-transit_{\alpha_1, C_\alpha}$,
 - iii. $snap(\alpha) = (s, in-transit_{\beta_1\beta_2, C_\beta})$, and
 - iv. the first state of α_3 is the first state of β_4 , and $in-transit_{\beta_1\beta_2s\beta_3, C_\beta} = in-transit_{\alpha_1\alpha_2, C_\alpha}$.

Condition (1) captures the idea that the computation of the application processes in the system with the snapshot algorithm should be a legal computation in the application system alone. Condition (2) concerns the snapshot itself. It says that if a snapshot is requested, then (a) snapshot information eventually is reported for each application process $u_i \in \mathcal{I}$, and (b) the snapshot produced must be a consistent recent state of the system. Note the similarity of part (b) to the informal statement of the global snapshot problem given at the beginning of this section, where the “initiation” state refers to the last states of α_1 and β_1 and the “termination” state refers to the first states of α_3 and β_4 .

In the specification of the global snapshot problem, we used the states of the underlying algorithm to express the recency condition. This would not have been possible with an ordinary schedule module specification, and points out the need for partial execution modules in conjunction with superposition.

4.2 The Algorithm

In this section, we use the I/O automaton model with superposition extensions to describe the global snapshot algorithm of Chandy and Lamport [2]. In the original paper by Chandy and Lamport, the snapshot algorithm is described being “superimposed” on top of the application program. However, the algorithm is actually presented as a modification to the underlying application program, and care is taken to ensure that the modification does not disrupt the application. Chandy and Misra [3] describe the algorithm in UNITY, and Nour [16] recasts that work using I/O automata. In both of these presentations, however, the distinctions between the snapshot algorithm and the underlying application become blurred. That is, the UNITY program resulting from superposing the snapshot algorithm on the application is monolithic; it does not preserve the essential separation of actions under the control of the snapshot protocol and the actions under the control of the application. This is partly due to the lack of separation of inputs and outputs, but is largely due to the absence of a mechanism for partitioning the actions of a program into separate processes. Using I/O automaton superposition, we are able to achieve a formal separation of the application program from the global snapshot protocol. The “built-in” partition of locally-controlled actions of an I/O automaton allows

us to model the actions of the snapshot protocol and the actions of the application as being under the control of *different* processes.

To model the global snapshot algorithm, we define a *snapshot automaton* p_i for each $i \in I$. In a sense, p_i encapsulates the corresponding application automaton u_i , acting as a buffer between the application processes and the network. Because of this structure, the SEND and the RCV actions of the application are no longer shared with the network, but are instead shared with the snapshot automaton. The snapshot automaton, in turn, interacts with the network using MSG_SEND and MSG_RCV actions to avoid naming conflicts. We postulate a renamed network, where the SEND actions are renamed to be MSG_SEND actions and the RCV actions are renamed to be MSG_RCV actions. In this way, the encapsulation structure is supported without renaming the actions of the application processes.

Each snapshot automaton p_i , $i \in I$ has several state components. The components *state-snapped* and *chan-snapped* $[j]$, $j \in I$ are boolean variables, initially false, that record whether or not the state of u_i and the states of the various “incoming channels” adjacent to u_i have been recorded for the snapshot. The components *in-queue* $[j]$ and *out-queue* $[j]$, for each $j \in I$, are queues of messages, initially empty. These contain messages that are waiting to be delivered to (or sent from) application process u_i . Recall that a message for process u_i from process u_j is not delivered directly to process u_i from the network. Instead, the message is delivered to the snapshot automaton p_i , which places the message in *in-queue* $[j]$ for later delivery. Similarly, when process u_i sends a message to process u_j , the SEND action is not shared with the network automaton. Instead, the snapshot automaton p_i puts the message in *out-queue* $[j]$ and later sends out the message. The component *snapshot*, initially undefined, takes on values in *states*(u_i), and is used to record the snapshot of the application’s state at u_i . Similarly, *chan-state* $[j]$, for $j \in I$ is an initially empty queue of messages that is used to record the state of the incoming channel from process j . Finally, the state component *done* is a boolean variable indicating whether or not p_i has reported the results of its local snapshot in a DONE $_i$ action.

The signature and transition relation are shown in Figure 1. In the code, if q is a queue, then *head*(q) refers to the first item in the queue (i.e., the one that would be dequeued next), or *nil* if the queue is empty. The function *tail*(q) refers to the queue that results from dequeuing *head*(q). If q has zero or one elements, then *tail*(q) = *nil*. The notation $q \circ a$ refers to the queue that results from enqueueing element a in the queue q . Since the snapshot automaton p_i is designed to be superposed on top of the application process automaton u_i , we use the state component *app* to refer to the state of the application process automaton u_i . One can easily check that p_i is unconstrained (in fact, completely unconstrained) for $\{app\}$. (See Lemma 11.) A step (s', π, s) appears in the transition relation for p_i iff the precondition for π holds in state s' , and state s is derived from s' according to the assignments in the effect of π . If no precondition is given, then it is assumed to be true in all states.

The partition *part*(p_i) is defined as follows. For each $j \in I$, there are two classes: all actions of the form RCV(m, i, j), $m \in \mathcal{M}$, and all actions of the form MSG_SEND(m, i, j), $m \in \mathcal{M}$. Finally the action DONE $_i$ is in a separate class.

The snapshot algorithm is initiated at p_i either by a START $_i$ action from the environment or by receipt of a marker message (#). The environment may generate any number of START messages for any number of snapshot processes. However, we view the first START message in an execution as the start of the global snapshot protocol. The first time a START $_i$ occurs or p_i receives a marker, p_i records the local state of the underlying application automaton u_i (and records the state of the incoming channel on which the marker was received as being empty), places a marker in the queue for all of its outgoing channels, and begins keeping track of all messages received on its incoming channels in the *chan-state* variables. Any later START $_i$ message is ignored. Any later receipt of

Input actions: START_i
 $\text{MSG_RECEIVE}(m, j, i), m \in \mathcal{M}, j \in \mathcal{I}$
 $\text{SEND}(m, i, j), m \in \mathcal{M}, j \in \mathcal{I}$

Output actions: $\text{RCV}(m, j, i), m \in \mathcal{M}, j \in \mathcal{I}$
 $\text{MSG_SEND}(m, i, j), m \in \mathcal{M}, j \in \mathcal{I}$
 $\text{DONE}_i(a, c), s \in \text{states}(u_i), c \in \text{chans}$

- START_i
 Effect: if $s'.\text{state-snapped} = \text{false}$ then
 $s.\text{snapshot} = s'.\text{app}$
 $s.\text{state-snapped} = \text{true}$
 $\forall k \in \text{out-chans}(i), s.\text{out-queue}[k] = s'.\text{out-queue}[k] \circ \#$
- $\text{MSG_RCV}(m, j, i)$
 Effect: if $m = \#$ then
 if $s'.\text{state-snapped} = \text{false}$ then
 $s.\text{snapshot} = s'.\text{app}$
 $s.\text{state-snapped} = \text{true}$
 $\forall k \in \text{out-chans}(i), s.\text{out-queue}[k] = s'.\text{out-queue}[k] \circ \#$
 $s.\text{chan-snapped}[j] = \text{true}$
 else
 $s.\text{in-queue}[j] = s'.\text{in-queue}[j] \circ m$
 if $s'.\text{state-snapped} = \text{true} \wedge s'.\text{chan-snapped}[j] = \text{false}$ then
 $s.\text{chan-state}[j] = s'.\text{chan-state}[j] \circ m$
- $\text{SEND}(m, i, j)$
 Effect: $s.\text{out-queue}[j] = s'.\text{out-queue}[j] \circ m$
- $\text{RCV}(m, j, i)$
 Precondition: $m = \text{head}(s'.\text{in-queue}[j])$
 Effect: $s.\text{in-queue}[j] = \text{tail}(s'.\text{in-queue}[j])$
- $\text{MSG_SEND}(m, i, j)$
 Precondition: $m = s'.\text{head}(\text{out-queue}[j])$
 Effect: $s.\text{out-queue}[j] = \text{tail}(s'.\text{out-queue}[j])$
- $\text{DONE}_i(a, c)$
 Precondition: $s'.\text{state-snapped} = \text{true}$
 $\forall j \in s'.\text{in-chans}, s'.\text{chan-snapped}[j] = \text{true}$
 $s'.\text{done} = \text{false}$
 $a = s'.\text{snapshot}$
 $\forall j \in \mathcal{I}, c(j) = s'.\text{chan-state}(j)$
 Effect: $s.\text{done} = \text{true}$

Figure 1: Global Snapshot Automaton p_i .

a marker on a given “channel” j causes p_i to set $chan_snapped[j]$ to true, which prevents p_j from adding later messages to $chan_state[j]$. Once the state of u_i and the states of all incoming channels have been snapped, p_i may issue a *DONE* action, reporting the snapshot information.

We now prove some simple properties of p_i . The compositionality properties of I/O automata will allow us to use these local results in proving properties of larger systems containing p_i .

Lemma 11: Automaton p_i is completely unconstrained for app .

Proof: By inspection of the code for p_i , app never appears in a precondition, and never appears on the left hand side of an assignment in an effect. ■

Lemma 12: Let α be a fair execution of p_i . Then there exists a live correspondence relation for SEND and MSG_SEND in α , and there exists a live correspondence relation for MSG_RCV and RCV in α .

Proof: By definition, a SEND(m, i, j) action for p_i places m into $out_queue[j]$. Only a MSG_SEND(m, i, j) action can remove m from that queue, and the only precondition for a MSG_SEND(m, i, j) action is that m is at the head of $out_queue[j]$. Therefore, since α is a fair execution, exactly one MSG_SEND(m, i, j) event eventually occurs for each element of $out_queue[j]$, in the order of the SEND events. This implies that there is a live correspondence relation for SEND and MSG_SEND in α . Similarly, by definition a MSG_RCV(m, j, i) action places m into $in_queue[j]$. Since α is a fair execution, exactly one RCV(m, j, i) event eventually occurs for each element of $in_queue[j]$, in the order of the MSG_RCV events. Therefore, there is a live correspondence relation for MSG_RCV and RCV in α . ■

The following lemma states several properties of executions of p_i .

Lemma 13: Let α be an execution of p_i containing states s' and s , in that order. Then $\forall j \in \mathcal{I}$,

1. If $s'.state_snapped = \text{true}$, then $s.state_snapped = \text{true}$.
2. If $s'.chan_snapped(j) = \text{true}$, then $s.chan_snapped(j) = \text{true}$.
3. If $s'.done = \text{true}$, then $s.done = \text{true}$.
4. If $s.chan_snapped(j) = \text{true}$, then $s.state_snapped = \text{true}$.
5. If $s'.state_snapped = \text{true}$, then $s.snapshot = s'.snapshot$.
6. If $s'.chan_snapped(j) = \text{true}$, then $s.chan_state(j) = s'.chan_state(j)$.
7. If $START_i$ occurs before state s , then $s.state_snapped = \text{true}$.
8. If $MSG_RCV(\#, j, i)$ occurs before state s , then $s.chan_snapped(j) = \text{true}$.
9. If $state_snapped = \text{false}$, then $chan_state(j) = \epsilon$.

Proof: Properties 1-3 are immediate from inspection of the code for p_i , since no action of p_i sets those variables to false. Property 4 follows from Property 1 and the definition of MSG_RCV, the only action that can set a $chan_snapped$ variable to true. Property 5 follows from the definitions of $START_i$ and MSG_RCV, which only modify $snapshot$ if $state_snapped$ is false. Similarly, Property 6 follows from the fact that a MSG_RCV action only modifies $chan_state(j)$ if $chan_snapped(j)$ is false. Property 7 follows from the definition of $START_i$ and Property 1. Property 8 follows from the definition of MSG_RCV and Property 2. Since $chan_state(j)$ is initially empty and only modified by a MSG_RCV action when $state_snapped = \text{true}$, Property 9 follows from Property 1. ■

In the following lemma, we use the above properties and invariants to show exactly what p_i reports in a DONE_i action.

Lemma 14: Let α be an execution of p_i containing a $\text{DONE}_i(a, c)$ action, and let s be the first state of α in which $\text{state-snapped} = \text{true}$. Then $a = s.\text{app}$, and for all $j \in \mathcal{I}$, $c(j)$ contains the sequence of messages m appearing in all the $\text{MSG_RCV}(m, j, i)$ actions between state s and the first state s' in which $\text{chan-snapped}(j) = \text{true}$.

Proof: From the precondition on DONE_i we know that state s must exist. The only actions that can cause state-snapped to become true are the START_i and MSG_RCV actions. When either of these actions sets $\text{state-snapped} = \text{true}$, they also copy app into snapshot . Therefore, $s.\text{snapshot} = s.\text{app}$. So, from Properties 1 and 5 of Lemma 13, we know that this value of snapshot remains fixed for the remainder of α . Therefore, by the definition of DONE_i , $a = s.\text{snapshot} = s.\text{app}$.

For all $j \in \mathcal{I}$, we know from Property 9 of Lemma 13 that in all states s' before s in α , $s'.\text{chan-state}[j] = \epsilon$. When the first $\text{RCV_MSG}(\#, j, i)$ occurs in α , $\text{chan-snapped}(j)$ is set to true, and $\text{chan-state}(j)$ becomes fixed by Properties 2 and 6. Then by the definition of MSG_RCV , the sequence of messages m appearing in all the $\text{MSG_RCV}(m, j, i)$ actions between state s and the first state s' in which $\text{chan-snapped}(j) = \text{true}$ is exactly the sequence of messages added to $\text{chan-state}(j)$ in α , and they are added in the order of occurrence in α . Therefore, by definition of DONE_i , the lemma holds. ■

4.3 Proof of Correctness

Throughout the proof, we use subscripts to distinguish the state components of different processes. For example, app_i refers to the app component of p_i .

Let automaton P be the composition of all p_i , $i \in \mathcal{I}$, and let X be the set of variables app_i , $i \in \mathcal{I}$. Let $Q = \mathcal{U}(P, X)$. We wish to show that Q solves partial execution module G . First, we prove a statement about interprocess communication in Q , and then turn directly to the main result.

Lemma 15: Let α be an admissible fair extended execution of Q . Then there exists a live correspondence relation for SEND and RCV in α .

Proof: Since α is admissible, there is a live correspondence relation between MSG_SEND and MSG_RCV in α . Therefore, by Lemmas 12 and 10, we have the desired result. ■

Theorem 16: Automaton Q solves partial execution module G .

Proof: The organization of the proof follows the definition of G . Let γ be an admissible fair extended execution of Q , and let $\alpha = \gamma[(\text{acts}(G), X)]$. For condition (1) of G , we wish to show that $\text{sched}(\alpha) \upharpoonright U \in \text{fairbehs}(S_{\text{app}})$. From the hypothesis (α admissible), we know that for all $i \in \mathcal{I}$, $\alpha \upharpoonright u_i \in \text{fairbehs}(u_i)$. Therefore, from standard I/O automaton compositionality results, we know that $\alpha \in \text{fairexecs}(U)$. And from Lemma 15, we know that $\text{sched}(\alpha)$ is network admissible. Therefore, $\text{sched}(\alpha) \upharpoonright U \in \text{fairbehs}(S_{\text{app}})$.

For condition (2) of G , suppose that a START action occurs in α . For (2a), we wish to show that exactly one DONE_i occurs in α for each $i \in \mathcal{I}$. For each $i \in \mathcal{I}$ when the first START_i or $\text{MSG_RCV}(\#, j, i)$, $j \in \mathcal{I}$, action occurs, a marker ($\#$) is placed into every $\text{out-queue}[j] \in \text{out-chans}(i)$. Therefore, if a START_i occurs or p_i receives a marker, then for all $j \in \text{out-chans}(i)$, a $\text{MSG_SEND}(\#, i, j)$ eventually occurs. Since α is admissible, a $\text{MSG_RCV}(\#, i, j)$ eventually occurs for all $j \neq i$. We know that the graph CHANS is strongly connected. Thus, if a START_i occurs in α , then eventually a marker is received by each p_i on each of its incoming channels. The first time a START_i occurs or a marker is received at p_i , the state of u_i is recorded. Furthermore, when a

$\text{MSG_RCV}(\#j, i)$ occurs, p_i records the state of incoming channel j . Therefore, it is eventually the case that for all $i, j \in \mathcal{I}$, $\text{state-snapped}_i = \text{true}$ and $\text{chan-snapped}[j] = \text{true}$. By Properties 1 and 2 of Lemma 13, we know that these are stable properties. Therefore, since the preconditions on DONE_i eventually become true and remain true until it occurs, a DONE_i action must eventually occur in α for each $i \in \mathcal{I}$. By Property 3 of Lemma 13 at most one DONE_i action can occur in α for each $i \in \mathcal{I}$, since that action sets done_i to true. This completes the proof of part (2a).

We prove part (2b) by a construction similar to the one presented by Chandy and Lamport [2]. For all $i \in \mathcal{I}$, let s_i^* be the first state in α in which $\text{state-snapped}_i = \text{true}$. (We have already shown that such a state must exist.) Now, for all $i \in \mathcal{I}$, mark all actions of $\alpha|u_i$ after s_i^* as *distinguished*. Note that all actions in α_1 are not distinguished, all actions in α_3 are distinguished, and α_2 contains a mixture of distinguished and undistinguished actions. We construct $\beta = \beta_1\beta_2s\beta_3\beta_4$ as follows:

1. Let $\beta_1 = \alpha_1|U$.
2. Let β_2 (β_3) contain the sequence of undistinguished (distinguished) actions of U in α_2 , where each action π is followed by state $s_\beta \in \text{states}(U)$ such that for all $i \in \mathcal{I}$,
 - (a) if $\pi \in \text{acts}(u_i)$ then $s_\beta|u_i = s_\alpha|u_i$, where s_α is the state following π in α , and
 - (b) if $\pi \notin \text{acts}(u_i)$ then $s_\beta|u_i = s'_\beta|u_i$, where s'_β is the previous state in β .
3. Let $\beta_4 = \alpha_3|U$.

Informally, we construct β from $\alpha|U$ by “delaying” the actions of a process that has recorded its local snapshot until all the remaining processes have also recorded theirs. The sequence β_1 is the prefix of $\alpha|U$ up to the first START action. The sequence β_2 contains all the remaining actions of $\alpha|U$ for processes that have not yet taken their local snapshots. The sequence β_3 contains all the “delayed” actions, up until the last process reports its snapshot. Finally, β_4 is the suffix of $\alpha|U$ after the global snapshot has been completed.

Clearly, for all $i \in \mathcal{I}$, $\beta|u_i = \alpha|u_i$. Therefore, from standard I/O automaton compositionality results, we know that β is an execution of U . Next, we need to show that there exists a live correspondence relation \mathcal{C}_β for SEND and RCV in β . We will show, in fact, that it is the *same* correspondence relation as in α . We know that the same actions occur in β as in α . Therefore, the only condition we need to show is that for each SEND action, the corresponding RCV occurs later in β :

Suppose (for contradiction) that there exists $\text{RCV}(m, i, j)$ in β such that the corresponding $\text{SEND}(m, i, j)$ occurs later in β . The only way this could happen is for the RCV to be an undistinguished action in α and the SEND to be a distinguished action in α , or else they could not have been reordered by the construction. However, if the SEND is a distinguished action, then the message from that SEND must be preceded in the outgoing channel by a marker message from u_i to u_j , so the RCV for the marker occurs at the u_j before the RCV for m , after which $\text{state-snapped}_j = \text{true}$. This means that any later actions of u_j are distinguished, a contradiction. Therefore, the live correspondence relation \mathcal{C}_β exists.

We now consider each of the four properties in condition (2b). Property (i) holds immediately from the construction, since the construction preserves the order of events at each automaton u_i , $i \in \mathcal{I}$. We know that α_1 is the prefix of α up to the first START action. Since no process p_i sets $\text{state-snapped}_i = \text{true}$ until after the first START action occurs, we know from the construction that $\alpha_1|U = \beta_1|U$. Therefore, the Property (ii) holds. Since $\beta_1\beta_2$ contains exactly the sequence of undistinguished actions in α , we know that for all $i \in \mathcal{I}$, $s|u_i$ is the state of u_i in α when state-snapped_i first becomes true. Moreover, we know that for all $i \in \mathcal{I}$, $\text{in-transit}_{\beta_1\beta_2, \mathcal{C}_\beta}(i)$ maps each

$j \in \mathcal{J}$ to the set of all messages sent by u_j before $state-snapped_j = \text{true}$, but not received by u_i before $state-snapped_i = \text{true}$. Whenever $state-snapped_j$ becomes true for the first time, p_j places a marker in all outgoing channels. Therefore, for all $i \in \mathcal{I}$, $\text{in-transit}_{\beta_1\beta_2, c_\beta}(i)$ maps each $j \in \mathcal{J}$ to exactly the sequence of messages m appearing in all the $\text{MSG_RCV}(m, j, i)$ of α between the first state in which $state-snapped_i = \text{true}$ and the last state before a marker message is received by p_i along the channel from p_j . Since receipt of a marker message from p_j results in $chan-snapped_i(j) = \text{true}$ (Property 8 of Lemma 13), we know from Lemma 14 that Property (iii) holds. From the construction $\beta_4 = \alpha_3|U$. Since the set of actions in $\alpha|U$ is exactly the set of actions in β and the correspondence relations for SEND and RCV are the same, Property (iv) holds. ■

5 Conclusion

In this paper, we have extended the I/O automaton model to permit superposition of program modules. This provides a unified model that permits one to reason locally about components of a distributed system, and to combine those modules through composition and/or superposition in such a way that the essential properties of the components are preserved in the resulting system.

Acknowledgments

I thank Nancy Lynch for many useful comments, Hagit Attiya and Nissim Francez for helpful discussions, and Ken Cox for his suggestions for improving the presentation.

References

- [1] Luc Bougé and Nissim Francez. A compositional approach to superimposition. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 240–249, January 1988.
- [2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [3] K. Mani Chandy and Jayadev Misra. *A Foundation of Parallel Program Design*. Addison-Wesley, Reading, MA, 1988.
- [4] A. Fekete, N. Lynch, Y. Mansour, and J. Spinelli. The data link layer: The impossibility of implementation in face of crashes. Technical Memo MIT/LCS/TM-355.b, MIT Laboratory for Computer Science, August 1989. Submitted for publication.
- [5] Limor Fix, Nissim Francez, and Orna Grumberg. Program composition and modular verification. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming, LNCS 510*, pages 93–114. Springer-Verlag, July 1991.
- [6] Nissim Francez and Ira R. Forman. Superimposition for interacting processes. In *Proceedings of the 1st International Conference on Concurrency Theory, Theories of Concurrency: Unification and Extension, LNCS 527*, pages 230–245. Springer-Verlag, August 1990.
- [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] Kenneth J. Goldman. A compositional model for layered distributed systems. In *Proceedings of the 2nd International Conference on Concurrency Theory, LNCS 527*, pages 220–234. Springer–Verlag, August 1991.
- [9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.
- [10] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the 4th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, August 1985.
- [11] Bengt Jonsson. Compositional specification and verification of distributed systems. Technical Report SICS/R-90/90010, Swedish Institute of Computer Science, October 1990.
- [12] Bengt Jonsson. Simulations between specification of distributed systems. In *Proceedings of the 2nd International Conference on Concurrency Theory, LNCS 527*, pages 346–360. Springer–Verlag, August 1991.
- [13] 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.
- [14] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3), 1989.
- [15] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [16] Magda F. Nour. An automata-theoretic model for UNITY. Technical Report MIT/LCS/TM-400, MIT Laboratory for Computer Science, June 1989. Senior Thesis.