An Automata-Theoretic Model for UNITY

by

Magda F. Nour

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

at the Massachusetts Institute of Technology

June 1989

Author_____

Department of Electrical Engineering and Computer Science
May 22, 1989

Certified by_____

Nancy A. Lynch
Thesis Supervisor

Accepted by_____

Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

An Automata-Theoretic Model for UNITY

by

Magda F. Nour

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1989

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Abstract

UNITY - Unbounded Nondeterministic Iterative Transformations - is a computational model and a proof system to aid in the design of parallel programs developed by K. Mani Chandy and Jayadev Misra at the University of Texas.

The Input/Output Automaton model is a computational model developed by Nancy Lynch and Mark Tuttle at MIT that may be used to model concurrent and distributed systems.

This thesis will connect these two theories. Specifically, it will

1. define UNITY Automata, a subset of I/O Automata based on the UNITY computational model, the UNITY program,
2. define a mapping from UNITY programs to UNITY Automata,
3. adapt the UNITY proof concepts to the I/O Automaton computational model in order to obtain UNITY style proof rules for I/O Automata,
4. adapt UNITY composition operators to the I/O Automaton model and obtain composition proof rules for them, and
5. consider various examples illustrating the above work.

In addition, this paper introduces an augmentation to the I/O Automaton model which facilitates reasoning about randomized algorithms, adapts UNITY concepts to it, and presents an example of a UNITY style high probability proof using such a model.

Thesis Supervisor: Nancy A. Lynch
Title: Professor, Department of Electrical Engineering and Computer Science

Contents

Acknowledgements

Nancy Lynch has been an inspiration to me as both an excellent professor and a prominent researcher. I am especially indebted to her for introducing me to this fascinating field and would like to thank her for her suggestions, encouragement, and patience as my thesis advisor.

Chapter 1

Introduction

In [CM], Chandy and Misra introduce a theory – a computational model and a proof system – called UNITY: Unbounded Nondeterministic Iterative Transformations. The computational model they use is that of a UNITY program which is comprised of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A program execution starts from any state satisfying the initial conditions and goes on forever. In each step of an execution some assignment statement is selected nondeterministically and executed. This nondeterministic selection is subject to a fairness constraint that requires that every statement be selected infinitely often. The goal of their book is to use UNITY to develop UNITY programs systematically for a variety of architectures and applications from a problem statement. Such a program is developed by first defining specifications that insure correctness in terms of conditions on variables, then these specifications are translated into a UNITY program. At that point the program may be further refined by adding more details to the specifications so that it can be mapped to some target architecture efficiently.

In [LT], Lynch and Tuttle introduce the Input/Output Automaton model, a computational model that can be used as a tool for modeling concurrent and distributed systems. It is somewhat like a traditional nondeterministic finite state automaton without the restriction of a finite set of states. An I/O automaton is defined by its initial states, states, actions, transition steps, and partition of local actions. It has three kinds of actions: input, output and internal. It generates output and internal actions autonomously and thus these are called local actions. Input actions, on the other hand, are generated by the environment and thus must be enabled at

every step. Every input and output action is instantaneously transmitted to the automaton or the environment respectively. An I/O automaton may be made up of any number of primitive components. The partition of local actions indicates those actions that may be thought of as under the control of some primitive system component and thus a simple notion of fairness is defined permitting each of the primitive components of the automaton to have infinitely many chances to perform a local action. I/O automata can be composed to yield other I/O automata. This composition operator connects each output action of one automaton with input actions of other automata resulting in a system where an output action is generated autonomously by exactly one component and instantaneously transmitted to those components with that action as an input action. Since input actions are always enabled, these actions are executed simultaneously with the output step.

This paper does not discuss the problem of program development but instead defines a direct mapping from a UNITY program to a special kind of I/O automaton called a UNITY automaton. A UNITY automaton is defined based on the concept of a UNITY program. This UNITY automaton may be seen as a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements just as the UNITY program is, however the UNITY automaton is described in the form of an I/O automaton. In the UNITY automaton, the variables are defined by the states of the automaton, the initial variable values are defined by the initial states, and the set of multiple-assignment statements are defined by the transition steps. Using this mapping, all of the program development steps described in [CM] may thus be used to develop UNITY automata. The reader is referred to [CM] for further discussion in that area.

In [CM] *properties* of a UNITY program are expressed in terms of predicates, such as "predicate P is always true", and are associated with the entire UNITY program. Most such properties are expressed using assertions of the form $\{p\}s\{q\}$, where s is universally or existentially quantified over the statements of the program. All properties of a UNITY program can be expressed directly using assertions. However, since it is cumbersome to use assertions all the time, in [CM] certain kinds of properties that arise often in practice have been given names and theorems about them have been derived. We will present analogous properties for I/O automata and will refer to them as UNITY properties. The UNITY approach of using these

properties to generate proofs about UNITY programs shall be referred to as UNITY proof concepts.

The focus of this paper is to adapt the UNITY proof concepts to I/O automata and to use these proof concepts to prove interesting properties for I/O automata. In addition to the proof system, the UNITY composition operators are adapted to I/O automata for the same purpose. Several basic examples are presented. Additionally an augmentation on the I/O automaton model is introduced to allow the straightforward representation of randomized algorithms. UNITY proof concepts are again adapted to this model and expanded to facilitate reasoning about high probability properties and probabilistic proofs. An example of the use of these proof concepts is presented.

Throughout this paper we shall assume that each I/O automaton, A, has a set of associated variables, $vars(A)$, that define its states. We shall also assume that each such variable, $v \in vars(A)$, has associated with it a set of initial values, I_v, and a range of possible values, X_v. The states of the I/O automaton are defined by the values of its associated variables. This definition can be made because any set of states and state transitions may be defined in terms of values of arbitrary variables. We shall use the expression $s(v)$ to denote the value of variable v at state s.

In chapter 2 UNITY proof concepts are described and put in terms of I/O automata properties. Several lemmas are stated about fair executions of I/O automata that satisfy certain UNITY properties. In the final section of Chapter 2, we extend some definitions of discrete state discrete transition Markov processes to apply to I/O automata. Chapter 3 links UNITY programs to UNITY automata. First, UNITY programs are described as they are defined in [CM], then UNITY automata are introduced, a mapping from UNITY programs to UNITY automata is defined, and the effects of such a mapping on the UNITY properties is discussed. In section 3.5, the composition operators for UNITY programs, Union and Superposition, are described as they are in [CM] and analogous composition operators are defined for UNITY automata. Chapter 4 expands on section 3.5 and defines Union and Superposition for general I/O automata, then reviews the composition operator defined in [LT] and finally expands upon that to define a new composition operator. Chapter 5 introduces a new kind of I/O automata called randomized I/O automata and adapts and expands UNITY proof concepts to reason about

them. The final chapter concludes the paper with some final remarks, possible applications and ideas for further research.

Chapter 2

UNITY Proof Concepts for I/O Automata

This chapter contains definitions of UNITY properties for I/O automata that correspond to those used for UNITY programs. The I/O automaton model, unlike the UNITY program model, classifies I/O automaton actions as either local or input. The input actions of the automaton must be enabled from every state. Due to this requirement, it is often interesting to make a distinction between properties of a general execution of an I/O automaton and properties of an execution consisting of only local actions. For this reason, in defining UNITY properties for I/O automata we define define both an analogous general UNITY property and an analogous "local" property.

Specifically, in this chapter we define the three fundamental logical UNITY relations: *unless*, *ensures*, and *leads to*. Also defined are two special cases of *unless*. *stable* and *invariant*, the notion of a *fixed point*, and two additional properties. The *unless* properties are *safety* properties and the *ensures* and *leads to* properties are progress properties. Lemmas are also presented relating satisfaction of some UNITY properties to fair execution characteristics.

2.1 UNITY-Style Assertions for I/O Automata

A UNITY program satisfies {p} w {q} if for any state of the program satisfying predicate p, after the execution of the statement w the state of the program satisfies q.

We define an analogous property of I/O Automata. An I/O Automaton A satisfies $\{P\}\pi\{Q\}$ where P and Q are sets of states in $states(A)$ if for all $(s, \pi, s') \in steps(A)$, $s \in P \Rightarrow s' \in Q$.

2.2 Unless

An I/O Automaton A satisfies P *unless* Q provided A satisfies $\{P \cap \sim Q\}\pi\{P \cup Q\}$ for all $\pi \in acts(A)$

Lemma 1 *If A satisfies* P unless Q *then in any execution* $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then either there exists* $k \geq i$ *such that* $s_j \in P \cap \sim Q$ *for all* j *such that* $i \leq j < k$ *and* $s_k \in Q$, *or* $s_j \in P \cap \sim Q$ *for all* $j \geq i$.

Proof: Consider the first occurrence of a state, s_k $(k \geq i)$ in α that is not in $P \cap \sim Q$. If there is no such state then for all $j \geq i$, $s_j \in P \cap \sim Q$ and the lemma holds. If $s_k \in Q$ then the lemma holds. Let us assume $s_k \notin Q$; then it must be in $\sim Q \cap \sim P$. However by the definition of *unless*, for all steps (s, π, s') in steps(A), $s \in P \cap \sim Q \Rightarrow s' \in P \cup Q$. We know that $s_{k-1} \in P \cap \sim Q$ by our definition of s_k so $s_k \in P \cup Q \Rightarrow s_k \notin \sim P \cap \sim Q$, a contradiction. Thus $s_k \in Q$ and the lemma holds. ■

An I/O Automaton A satisfies P *local-unless* Q provided A satisfies $\{P \cap \sim Q\}\pi\{P \cup Q\}$ for all $\pi \in local(A)$

Lemma 2 *If A satisfies* P local-unless Q *then in any execution* $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then either (a) there exists* $k \geq i$ *such that* $s_j \in P \cap \sim Q$ *for all* $(i \leq j < k)$ *and* $s_k \in Q$, *(b) there exists* $k > i$ *such that* $s_j \in P \cap \sim Q$ *for all* $(i \leq j < k)$ *and* $\pi_k \notin local(A)$, *or (c)* $s_j \in P \cap \sim Q$ *for all* $(j \geq i)$.

Proof: Consider the first occurrence of a state, s_k $(k \geq i)$ in α that is not in $P \cap \sim Q$. If there is no such state then for all $j \geq i$, $s_j \in P \cap \sim Q$ (case (c)) and the lemma holds. If $s_k \in Q$ (case (a)) then the lemma holds. Otherwise, $s_k \notin Q$, let us assume this is the case; then s_k must be in $\sim Q \cap \sim P$, thus $k > i$ since we know $s_i \in P$. By the definition of $local - unless$, for all steps (s, π, s') in $steps(A)$ where $\pi \in local(A)$, $s \in P \cap \sim Q \Rightarrow s' \in P \cup Q$. We know that $s_{k-1} \in P \cap \sim Q$ by our definition of s_k so $s_k \in P \cup Q \Rightarrow s_k \notin \sim P \cap \sim Q$, therefore $\pi_k \notin local(A)$ and the lemma holds. ■

Lemma 3 *If A satisfies* P unless Q *then A satisfies* P local-unless Q.

2.3 Ensures

A satisfies P *ensures* Q iff A satisfies P *unless* Q and there exists a class $C \in part(A)$ such that $\forall \pi \in C$, A satisfies $\{P \cap \sim Q\}\pi\{Q\}$ and for all states $s \in P \cap \sim Q$ some $\pi \in C$ is enabled (i.e. there exists a step (s, π, s') such that $\pi \in C$).

Lemma 4 *If A satisfies* P ensures Q, *then in any fair execution* $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then there is an index* $k \geq i$ *such that* $s_k \in Q$ *and* $\forall j, (i \leq j < k), s_j \in P \cap \sim Q$.

Proof: If s_i is also in Q then $i = k$ and the lemma is true, otherwise if s_i is in P but not in Q, then since A satisfies P *unless* Q, by Lemma 1 we know that if there is a state, $s_k \in Q, (k \geq i)$ where $s_j \notin Q$ for all $k > j \geq i$ (i.e. s_k is the first state in Q after s_i), then for all $k > j \geq i$, $s_j \in P \cap \sim Q$. Now we must show that such a state must necessarily exist. We know that there exists an class C of local actions of A such that for all states in $P \cap \sim Q$ any action in C will yield a new state that satisfies Q. We also know that in every state in $P \cap \sim Q$ there is an action in C enabled.

Case 1: α *is finite.* The fairness condition states that if α is a finite execution, then no action of C is enabled from the final state of α. Therefore α may not end in a state in $P \cap \sim Q$. Thus by Lemma 1 we know that since s_j cannot be in $P \cap \sim Q$ for all $j \geq i$ then there must exist some $k \geq i$ such that $s_k \in Q$ and for all $i \leq j < k$, $s_j \in P \cap \sim Q$ thus the lemma holds.

Case 2: α *is infinite.* By Lemma 1 we know that if it is not the case that $s_j \in P \cap \sim Q$ for all $j \geq i$ then the lemma holds. The fairness condition states that if α is an infinite execution, then either actions from C appear infinitely often in α, or states from which no action of C is enabled appear infinitely often in α. If the former is the case, then we know that an action from C must be chosen after a finite number of steps if the states of α continue to be in $P \cap \sim Q$. Let us consider the first such action $\pi_m \in C$, $(m \geq i)$ in α. We know that $s_{m-1} \in P \cap \sim Q$ by the definition of π_m, so by the definition of *ensures* s_m must satisfy Q and thus the lemma holds. If the other fairness condition holds, then states from which no action of C is enabled appear infinitely often in α. If this is the case, then there must be some state s_m $m > i$ such that no

action of C is enabled which means that s_m is not in $P \cap \sim Q$, and so by Lemma 1 the lemma holds. ■

A satisfies P *local-ensures* Q iff A satisfies P *local-unless* Q and there exists a class $C \in part(A)$ such that $\forall \pi \in C$, A satisfies $\{P \cap \sim Q\}\pi\{Q\}$.

Lemma 5 *If A satisfies P* local-ensures *Q, then in any fair execution* $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then either (a) there is an index* $k \geq i$ *such that* $s_k \in Q$ *and for all* $j, (i \leq j < k), s_j \in P \cap \sim Q$ *or (b) there is an index* $k > i$ *such that* $\pi_k \notin local(A)$ *and for all* $j, (i \leq j < k), s_j \in P \cap \sim Q$.

Proof: If s_i is also in Q then $i = k$ and the lemma is true, otherwise:

Case 1: α *is finite.* The fairness condition states that if α is a finite execution, then no action of C is enabled from the final state of α. Therefore α may not end in a state in $P \cap \sim Q$. Thus by Lemma 2 we know that since s_j cannot be in $P \cap \sim Q$ for all $j \geq i$ then there must exist either some $k \geq i$ such that $s_k \in Q$ and for all $i \leq j < k$, $s_j \in P \cap \sim Q$ or some $k > i$ such that $\pi_k \notin local(A)$ and for all $i \leq j < k$, $s_j \in P \cap \sim Q$ and thus the lemma holds.

Case 2: α *is infinite.* By Lemma 2 we know that if it is not the case that $s_j \in P \cap \sim Q$ for all $j \geq i$ then the lemma holds. The fairness condition states that if α is an infinite execution, then either actions from C appear infinitely often in α, or states from which no action of C is enabled appear infinitely often in α. If the former is the case, then we know that an action from C must be chosen after a finite number of steps if the states of α continue to be in $P \cap \sim Q$. Let us consider the first such action $\pi_m \in C (m \geq i)$ in α. We know that $s_{m-1} \in P \cap \sim Q$ by the definition of π_m, so by the definition of *ensures*, s_m must satisfy Q and thus the lemma holds. If the other fairness condition holds, then states from which no action of C is enabled appear infinitely often in α. If this is the case, then there must be some state s_m $m > i$ such that no action of C is enabled which means that s_m is not in $P \cap \sim Q$, and so the lemma holds. ■

Lemma 6 *If A satisfies* P ensures Q *then A satisfies* P local-ensures Q.

2.4 Leads To

A satisfies $P \mapsto Q$ (pronounced P *leads to* Q) iff there exists a sequence $P_1, \ldots P_k$, $(k \geq 2)$ of sets where $P_1 = P$, $P_k = Q$, and A satisfies P_i *ensures* P_{i+1} for all $1 \leq i \leq k - 1$.

Lemma 7 *If A satisfies $P \mapsto Q$ then in a fair execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A, if $s_i \in P$ then there exists an $m \geq i$, such that $s_m \in Q$.*

Proof: From the Lemma 4 we can say that in a fair execution of A, if P *ensures* Q, then the lemma holds. Otherwise, if $((P \text{ ensures } P_2) \wedge (P_2 \text{ ensures } P_3) \wedge \cdots \wedge (P_{k-1} \text{ ensures } Q))$, then by the same lemma we know that if $s_i \in P = P_1$ then there exists a $j_2 \geq i$ such that $s_{j_2} \in P_2 \Rightarrow \exists j_3 \geq j_2$ such that $s_{j_3} \in P_3 \Rightarrow \ldots \Rightarrow \exists j_k \geq j_{k-1}$ such that $s_{j_k} \in P_k = Q$, and the lemma holds. ■

A satisfies $P \overset{\ell}{\mapsto} Q$ (pronounced *P local leads to Q*) iff there exists a sequence $P_1, \ldots P_k$, $(k \geq 2)$ of predicates where $P_1 = P$, $P_k = Q$, and A satisfies P_i *local-ensures* P_{i+1} for all $1 \leq i \leq k-1$.

Lemma 8 *If A satisfies $P \overset{\ell}{\mapsto} Q$ then in a fair execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A, if $s_i \in P$ then there exists an $m \geq i$, such that $s_m \in Q$ or there exists an $m > i$ such that $\pi_m \notin local(A)$.*

Proof: Analogous to proof of Lemma 7. ■

Lemma 9 *If A satisfies $P \mapsto Q$ then A satisfies $P \overset{\ell}{\mapsto} Q$.*

Let us define further a specific path-defined leads to. In other words, instead of only specifying that there exists a sequence of $P_1, \ldots P_k (k \geq 1)$ of predicates such that A satisfies P_i *ensures* P_{i+1} for all $1 \leq i \leq k-1$, the intermediate predicates are listed. Therefore if it is known that A satisfies *P ensures Q*, *Q ensures R*, and *R ensures S* then we can write $P \overset{QR}{\longmapsto} S$ (pronounced P leads to S through QR). This is a useful property to talk about when examining the properties preserved in composition.

2.5 Fixed Point

A *fixed point* of an I/O Automaton is a state from which any action taken will leave the state unchanged. Note that fixed point predicates are not UNITY properties.

A *fixed point* of an I/O Automaton A is a state s such that for all steps (s, π, s'), s=s'.

A *local fixed point* of an I/O Automaton A is a state s such that for all steps (s, π, s'), s=s' if $\pi \in local(A)$.

Lemma 10 *If A has a* fixed point *s, then in a fair execution* $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ *of A, if* $s_i = s$ *then* $s_j = s$ *for all* $j \geq i$.

Proof: We will prove $s_j = s$ by induction on j. *Basis : j=i:* by definition of s_i, $s_j = s_i = s$.

Induction Hypothesis : by the definition of *fixed point*, for any step $(s_j, \pi_{j+1}, s_{j+1}) \in steps(A)$, $s_j = s \Rightarrow s_{j+1} = s_j = s$. ■

Lemma 11 *If A has a* local fixed point *s, then in a fair execution* $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ *of A, if* $s_i = s$ *then either* $s_j = s$ *for all* $j \geq i$ *or there exists a* $k > i$ *such that* $\pi_k \notin local(A)$ *and* $s_j = s$ *for all* $k > j \geq i$.

Proof: Analogous to proof of previous lemma ■

Lemma 12 *If s is a* fixed point *of A then it is also a* local fixed point *of A.*

The following properties are useful when discussing some compositions of I/O Automata with fixed points.

A *variable set fixed point* of a set of variables, V_{FP} in I/O Automaton A is a set of states in $states(A)$ such that for all steps $(s, \pi, s') \in steps(A)$, $s(v) = s'(v)$ for all $v \in V_{FP}$.

A *variable set local fixed point* of a set of variables, V_{FP} in I/O Automaton A is a set of states in $states(A)$ such that for all steps $(s, \pi, s') \in steps(A)$ where $\pi \in local(A)$, $s(v) = s'(v)$ for all $v \in V_{FP}$.

2.6 Detects

A satisfies P *detects* Q iff $P \subseteq Q$ and $Q \mapsto P$.

A satisfies P *local-detects* Q iff $P \subseteq Q$ and $Q \overset{\ell}{\mapsto} P$.

Lemma 13 *If A satisfies* P detects Q *then A satisfies* P local-detects Q.

2.7 Until

A satisfies P *until* Q iff P *unless* Q and $P \mapsto Q$.

A satisfies P *local-until* Q iff P *local-unless* Q and $P \overset{\ell}{\mapsto} Q$.

Lemma 14 *If A satisfies* P until Q *and then A satisfies* P local-until Q.

2.8 Stable Property

Define P to be *stable* in A if A satisfies P *unless* false.

Lemma 15 *In a fair execution $\alpha = s_0\pi_0 s_1\pi_1 \ldots$ of A where P is stable, if $s_i \in P$ then $s_j \in P$, for all $j \geq i$.*

Proof: By Lemma 1 if $s_i \in P$ then $s_j \in P$ for all $j \geq i$ or there exists some $k \geq i$ such that s_k satisfies false. Since we know no state satisfies false, then we know that the former holds and the lemma is true. ■

Define P to be *locally stable* in A if A satisfies P *local-unless* false.

Lemma 16 *In a fair execution $\alpha = s_0\pi_0 s_1\pi_1 \ldots$ of A where P is locally stable, if $s_i \in P$ then either $s_j \in P$, for all $j \geq i$ or there exists $k > i$ where $s_j \in P$ for all $i \leq j < k$ and $\pi_{k-1} \notin local(A)$.*

Proof: By Lemma 2 if $s_i \in P$ then either $s_j \in P$ for all $j \geq i$ or there exists some $k \geq i$ such that s_k satisfies false or $\pi_{k-1} \notin local(A)$. Since we know no state satisfies false, then we know that either $s_j \in P$ for all $j \geq i$ or there exists some $k \geq i$ such that $\pi_{k-1} \notin local(A)$ and the lemma is true. ■

Lemma 17 *If P is* stable *in A then P is also* locally-stable *in A.*

2.9 Invariant

P is invariant in A iff $start(A) \subseteq P$ and P is stable.

Lemma 18 *If P is invariant in A then in any execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A, $s_i \in P$ for all i.*

Proof: In α, we know that $\forall s_0 \in start(A), s_0 \in P$ because $start(A) \subseteq P$. We also know by Lemma 15 that if P is *stable* and $s_i \in P$ then $s_j \in P$ $\forall j \geq i$ and so the lemma holds. ■

Note that the converse of this lemma is not true. This is because the stability of P is defined by P *unless false*, which requires that $\forall (s, \pi, s') \in steps(A), s \in P \Rightarrow s' \in P$. The following

example illustrates the problem. In all reachable states of this automaton, $x > 0$, however it is not an *invariant* of the automaton.

states(A) = $\{(x, allow - decrement) : x \in \aleph, allow - decrement \in \{true, false\}\}$

start(A) = (2,true)

acts(A) = out(A) = $\{\pi_{decr}, \pi_{incr}\}$

steps(A) = $\{\{((x, true), \pi_{decr}, (x', false)) : x' = x - 1\}$
$\cup\{((x, allow - decrement), \pi_{incr}, (x', true)) : x' = x + 1\}\}$

part(A) = $\{\{\pi_{incr}, \pi_{decr}\}\}$

The key here is that π_{decr} is only enabled when $allow - decrement$ is true which is only the case after at least one increment of x (i.e. one execution of π_{incr}), therefore it is easy to see that x is never less than one in any reachable state of A, however, $x > 0$ is not an invariant because $x > 0$ is not stable since there is a step in $steps(A)$, namely $((1, true), \pi_{decr}, (0, false))$ that makes a transition from those states where $x > 0$ to those states where $x \leq 0$.

P is local invariant in A iff $start(A) \subseteq P$ and P is locally stable.

Lemma 19 *If P is* local invariant *in A then in any execution* $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ *of A, either* $s_i \in P$ *for all i or there exists a* $k > 0$ *such that* $\pi_k \notin local(A)$ *and* $s_j \in P$ *for all* $0 \leq j < k$.

Proof: In α, we know that $\forall s_0 \in start(A), s_0 \in P$ because $start(A) \subseteq P$. We also know by the previous lemma that if P is *locally stable* and $s_i \in P$ then either $s_j \in P$ for all $j \geq i$ or there exists a $k > i$ such that $\pi_k \notin local(A)$ in which case $s_j \in P$ for all $i \leq j < k$ and so the lemma holds. ■

Lemma 20 *If P is* invariant *in A then P is also* locally-invariant *in A.*

A state s' is *reachable* from a state s (denoted by $s \rightsquigarrow s'$) in A iff there exists a finite number of (or possibly zero) steps in steps(A) that can be taken from state s to state s'.

A state s' is *locally reachable* from a state s (denoted by $s \overset{\ell}{\rightsquigarrow} s'$) in A iff there exists a finite set of local steps in steps(A) that can be taken from state s to state s'.

Lemma 21 *If a state* s' *is* locally reachable *from a state s in A then* s' *is also* reachable *from s in A.*

Recall the discussion about the converse of Lemma 18. It is not true that if for all states of any execution of A some property holds that this property is an *invariant* of A. We would like to define a property for which this would be true as well as its converse. We call this property *virtually invariant.*

P is *virtually invariant* in A iff for all states $s \in start(A)$ if $s \rightsquigarrow s'$ then $s' \in P$.

Lemma 22 *If P is* invariant *then it is* virtually invariant.

Proof: The first clause of the definition of virtually invariant is the same as that of the invariant definition. By the definition of stable, we know for all steps (s, π, s') in steps(A), if $s \in P$ then $s' \in P$. By the definition of $s \rightsquigarrow s'$, if (s, π, s') is in steps(A) then $s \rightsquigarrow s'$ and by the definition of virtually invariant s' must be in P thus the lemma holds. ■

Note that the converse of Lemma 22 is not true.

Lemma 23 *P is* virtually invariant *in A iff in any execution $\alpha = s_0\pi_0s_1\pi_1\ldots$ of A, $s_i \in P$ for all i.*

Proof: If P is *virtually invariant* in A then in any execution $\alpha = s_0\pi_0s_1\pi_1\ldots$ of A, $s_i \in P$ for all i follows from Lemmas 18 and 22.

If in all executions $\alpha = s_0\pi_0s_1\pi_1\ldots$ of A, $s_i \in P$ for all i, then P is *virtually invariant* follows from the following argument. If in all executions α, $s_0 \in P$ then $start(A) \subseteq P$. If in all executions $s_i \in P$ for all i then for all s' such that $s \rightsquigarrow s'$ and $s \in start(A)$, $s' \in P$, thus P is *virtually invariant.* ■

P is *virtually locally invariant* in A iff $start(A) \subseteq P$ and for all states $s \in start(A)$ if $s \overset{\ell}{\rightsquigarrow} s'$ then $s' \in P$.

Lemma 24 *If P is* locally invariant *then it is* virtually locally invariant.

Proof: Analogous to the proof of the previous lemma. ■

Note that the converse of Lemma 24 is not true.

Lemma 25 *P is* virtually locally invariant *in A iff in any execution $\alpha = s_0\pi_0s_1\pi_1\ldots$ of A, either $s_i \in P$ for all i or there exists a $k \geq 1$ such that $\pi_{k-1} \notin local(A)$ and $s_j \in P$ for all $0 \leq j < k$.*

Proof: Analogous to proof of Lemma 23. ■

Lemma 26 *If P is* virtually invariant *then it is* virtually locally invariant.

2.10 State Transition Concepts

We shall now define some properties of I/O automata that are analogous to those of discrete-state discrete-transition Markov processes. We shall also extend these to discuss properties about sets of states rather than states and transitions from sets of states rather than from one state, specifically, we shall relate these to UNITY properties.

Here we shall classify properties as sets of states and consider the concept of limiting-state probabilities with respect to progress.

A *transient* state, s, is a state which in any execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A, if $s_i = s$ and $\exists j > i$ such that $s_j \neq s$ then $\forall k \geq j$, $s_k \neq s$. In other words if s was the state at some point in the execution, but did not continue to be, then s will not be the state at any later state of the execution. It is clear that there either must be some distinction between the states s_h where $h < i$ and s_j $j > i$ where $s_k = s$ for all $i \leq k < j$, if such states exist, otherwise, there would be the possibility of repeating the step in $steps(A)$ that lead to state s. We shall refer to this distinction as the state s *has not held* and s *has held* respectively.

A *recurrent* state, s, is one which in any execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A, if $s_i = s$ and $\exists j > i$ such that $s_j \neq s$ then there also exists a $k > j$ such that $s_k = s$. We may restate this as $s' \rightsquigarrow s$ for all s'.

A *state chain*, $s_1, s_2, s_3, ..., s_n$, is a chain of states such that for all $(s, \pi, s') \in steps(A)$, $s = s_i \Rightarrow s' = s_{i+1}$

We now define analogous traits for properties.

A *transient* property, P, is a property which in any execution $\alpha = s_0\pi_0 s_1\pi_1 \ldots$ of A, if $s_i \in P$ and $\exists j > i$ such that $s_j \notin P$ then $\forall k \geq j$, $s_k \notin P$. In other words if P held at some point in the execution, but did not continue to hold, then P will never hold at any later state of the execution. It is clear that there either must be some distinction between the states s_h where $h < i$ and s_j $j > i$ where $s_k \in P$ for all $i \leq k < j$, if such states exist, otherwise, there would be the possibility of repeating the step in $steps(A)$ that lead to property P. We shall

refer to this distinction as the property P *has not held* and P *has held* respectively. Thus, we may put this definition in UNITY terms by stating that P *has held* $\cap \sim P$ is stable.

A *recurrent* property, P, is one which in any execution $\alpha = s_0\pi_0 s_1\pi_1 \ldots$ of A, if $s_i \in P$ and $\exists j > i$ such that $s_j \notin P$ then there also exists a $k > j$ such that $s_k \in P$. This definition may be put in UNITY terms by stating $\sim P \mapsto P$.

A *property chain*, $P_1, P_2, P_3, ..., P_n$, is a chain of distinct properties such that P_i *ensures* P_{i+1}, or in other words $P_i \overset{P_{i+1}P_{i+2}...P_{j-1}}{\longmapsto} P_j$, for all $1 \leq i < j \leq n$.

Some interesting properties of an I/O automaton can be described in the above terms. I state the following lemmas as examples.

Lemma 27 *If there exists a* property chain *of A,* $P_1, P_2, P_3, ..., P_n$, *such that* P_n unless P_1 *then* $P_1 \cup P_2 \cup ... \cup P_n$ *is stable.*

Lemma 28 *If there exists a* property chain *of A,* $P_1, P_2, P_3, ..., P_n$, *such that* P_n ensures P_1 *then* P_i *is a* recurrent *property for all* $1 \leq i \leq n$.

Lemma 29 *If there exists a* property chain *of A,* $P_1, P_2, P_3, ..., P_n$, *such that* $P_n \mapsto Q$ *where* Q *is* stable *and* $Q \subseteq \sim P_m$ *then* P_i *is* transient *for all* $1 \leq i \leq m$.

We shall refer to some of these terms in Chapter 5.

Chapter 3

UNITY Programs and UNITY Automata

In this chapter we present the definition of a UNITY program as defined in [CM] and the the definition of a UNITY automaton, a special kind of I/O automaton which has the same characteristics and properties as a UNITY program. A mapping from UNITY programs to UNITY automata is also defined and the effects discussed. Later in this chapter UNITY program composition operators are described and corresponding UNITY automata composition operators are defined.

3.1 UNITY Program

A UNITY Program consists of

- a set V of *variables*, declared in the declare section of P,
- for each $v \in V$, a set X_v of *values* for v, determined by the domain of the type of v, declared in the declare section of P,
- for each $v \in$ V, a subset I_v of X_v of *initial values* for v, indicated in the initially-section of P (those variables not constrained in the initially section have $I_v = X_v$).
- a finite set W of *assignments* in the assign-section of P; each assignment, w, modifies a nonempty subset of V, V_w; each assignment, w, sets each variable in V_w to a function of

the values of the variables in V, so each w in W may be represented as a pair $(V_w, \{f^v : v \in V_w\})$ where f^v is a function from $\prod_{u \in V} X_u$ to X_v.

In the case of conditional assignments, there is a condition C. The conditional assignment may still be represented as a pair $(V_w, \{f^v : v \in V_w\})$. In this case, f^v is a function from $\prod_{u \in V} X_u$ to X_v such that for all states $s \notin$ C, $f^v(s) = s$.

UNITY programs also have *always* section assignments, each of which define a set of variables as functions of other variables in the program. The variables defined in this section are called *transparent* variables.

3.2 UNITY Automata

A Unity automaton is an I/O automaton A satisfying the following conditions.

1. There exists a set vars(A) of *variables*, where each v in vars(A) has an associated set X_v of *values* and an associated set I_v of *initial values*, where I_v is a subset of X_v.
2. All actions of A are output actions.
3. The set states(A) = $\prod_{v \in vars(A)} X_v$.
4. The set start(A) = $\prod_{v \in vars(A)} I_v$.
5. For all states s and all actions π there is a unique state s' having (s, π, s') in steps(A).
6. part(A) = $\{\{\pi\} : \pi \in acts(A)\}$

This UNITY automaton is a restricted I/O Automaton with only output actions, state-deterministic actions, cartesian product states, and singleton partition classes.

3.3 Mapping from UNITY Programs to Unity Automata

At this point we define a mapping from a UNITY program, P = (V_p, W_p), to a Unity Automaton, A = (vars(A), states(A), start(A), sig(A), steps(A), part(A)).

- vars(A) = V_p

- states(A) = $\prod_{v \in vars(A)} X_v$,
- start(A) = $\prod_{v \in vars(A)} I_v$,
- sig(A) = (in(A), out(A), int(A))

  where

  - out(A) = $\{\pi_w : w \in W\}$ and
  - int(A) = in(A) = $\emptyset$,

- steps(A) = $\{(s, \pi_w, s') : s'(v) = s(v)$ if $v \notin V_w$, otherwise $s'(v) = f_v(s)$ for $v \in V_w\}$,
- part(A) = $\{\{\pi\} : \pi \in$ acts(A)$\}$

Theorem 30 *The mapping as defined above of a UNITY program P yields a Unity Automaton.*

Proof: The yielded Automaton satisfies all the Unity Automaton conditions stated in the previous section. ■

Note that for all steps s ∈ steps(A) all statements in the always-section of P are represented as implicit variable definitions for all states, although the actual variables defined in the always section do not have components in the state vector. For example if x is always twice y then there need not be both x and y in the state vector because for each x there can only be one value of y. These variables are referred to in [CM] as *transparent variables* and will not be treated explicitly in the UNITY automata.

3.4 The Effect of Automaton Mapping on UNITY Properties

All of the UNITY properties for UNITY programs are directly related to those of UNITY automata, furthermore if a UNITY property holds for a UNITY program then it must hold for the UNITY Automaton that it maps to. This is easy to see since a predicate in UNITY is a predicate on the values of the variables of the program which is directly analogous to the state of the UNITY automaton. The execution of an assignment statement in a UNITY program is directly analogous to the execution of a step in the corresponding UNITY automaton, and because of the direct mapping from a UNITY program assignment to a UNITY automaton

step, the subsequent change of state is exactly the same. All of the definitions from Chapter 2 are exactly analogous to their UNITY counterparts in the same way and thus those properties that hold for a UNITY program hold for its corresponding UNITY automaton.

Theorem 31 *If a UNITY program P satisfies* p unless q *then the UNITY Automaton A = Automaton(P) satisfies* P unless Q *where P is the set of states in states(A) that satisfy predicate p and Q is the set of states in states(A) that satisfy predicate q.*

Theorem 32 *If a UNITY program P satisfies* p ensures q *then the UNITY Automaton A = Automaton(P) satisfies* P ensures Q *where P is the set of states in states(A) that satisfy predicate p and Q is the set of states in states(A) that satisfy predicate q.*

Theorem 33 *If a UNITY program P satisfies* $p \mapsto q$ *then the UNITY Automaton A = Automaton(P) satisfies* $P \mapsto Q$ *where P is the set of states in states(A) that satisfy predicate p and Q is the set of states in states(A) that satisfy predicate q.*

Theorem 34 *If a UNITY program P has a Fixed Point,* s_p*, then the UNITY Automaton A = Automaton(P) has a Fixed Point,* $s \in states(A)$ *which is its analogous state in A.*

Theorem 35 *If a UNITY program P satisfies* p *is* stable *then the UNITY Automaton A = Automaton(P) satisfies* P *is* stable *where P is the set of states in states(A) that satisfy predicate p.*

Theorem 36 *If a UNITY program P satisfies* p *is* invariant *then the UNITY Automaton A = Automaton(P) satisfies* P *is* invariant *where P is the set of states in states(A) that satisfy predicate p.*

Theorems 31 through 36 are true by the definition of the mapping.

3.5 UNITY Composition

UNITY has two forms of composition: composition by Union (denoted by the operator $[]$), and composition by Superposition (which we shall denote by $\triangleright$ with the lower level program on the right). The composition by Union is the more intuitive composition which shall be defined first.

3.5.1 Union

We shall investigate the [CM] definition of Union of two UNITY Programs P_1 and P_2. It is the same as appending their codes together. Union is a commutative, associative operator on programs.

In order for two UNITY programs P_1 and P_2 to be compatible they must satisfy the following requirements:

$\forall v \in V_1 \cap V_2$, $X_{v_1} = X_{v_2}$,

$\forall v \in V_1 \cap V_2$, $I_{v_1} = I_{v_2}$,

The UNITY definition of the union of UNITY programs P_1 and P_2 yields a UNITY program, $P = (V, W)$, where:

$V = V_1 \cup V_2$ and

$W = W_1 \cup W_2$

We shall now define such a Union operator for UNITY Automata.

UNITY Automata A_1 and A_2 must satisfy the following compatibility requirements to be combined using the Union operator:

1. X_v must be the same in A_1 and A_2 for all variables v in $vars(A_1) \cap vars(A_2)$,

2. I_v must be the same in A_1 and A_2 for all variables v in $vars(A_1) \cap vars(A_2)$, and

3. $acts(A_1) \cap acts(A_2) = \emptyset$

The Union of compatible UNITY Automata A_1 = (vars(A_1), states(A_1), start(A_1), sig(A_1), steps(A_1), part(A_1)) and A_2 = (vars(A_2), states(A_2), start(A_2), sig(A_2), steps(A_2), and part(A_2)) yields A = (vars(A), states(A), start(A), sig(A), steps(A), and part(A)), where:

- vars(A) = $vars(A_1) \cup vars(A_2)$,
- out(A) = out(A_1) $\cup$ out(A_2)
- in(A) = int(A) = $\emptyset$
- states(A) = $\prod_{v \in vars(A)} X_v$
- start(A) = $\prod_{v \in vars(A)} I_v$

- steps(A) = $\{(s, \pi, s')$: if $\pi \in acts(A_i)$, $(s_i, \pi, s_i') \in steps(A_i)$, and $s(v) = s_i(v)$ for all $v \in vars(A_i)$ then $s'(v) = s_i'(v)$ for all $v \in vars(A_i)$ and $s'(v) = s(v)$ for all $v \notin vars(A_i)\}$

- part(A) = $\{\{\pi\} : \pi \in$ acts(A)$\}$

The Union of two UNITY Programs as described earlier would yield a UNITY Program P_{comp}. The mapping of this P_{comp} would yield a UNITY Automaton, A_{comp}, defined as:

- vars(A_{comp}) = V_{comp}

- states(A_{comp}) = $\prod_{v \in V_{comp}} X_v$,

- start(A_{comp}) = $\prod_{v \in V_{comp}} I_v$,

- out(A_{comp}) = $\{\pi_w$: w $\in W_{comp}\}$
  = out(A_1) $\cup$ out(A_2)

- steps(A) = $\{(s, \pi_w, s') : s'(v) = s(v)$ if $v \notin V_w$,
  $s'(v) = f^v(s)$ if $v \in V_w$, for all $w \in W_{comp}\}$,

- part(A) = $\{\{\pi\} : \pi \in$ acts(A)$\}$

Theorem 37 *For any two UNITY programs P_1 and P_2, map($P_1 \parallel P_2$) = map(P_1) $\parallel$ map(P_2).*

Proof: We have just shown the result of map($P_1 \parallel P_2$). In section 3.3 we show the result of a mapping from a UNITY program P to a UNITY automaton A. Here we apply the Union operator to the mapping of P_1, called A_1, and the mapping of P_2, called A_2. The resulting automaton, A_{union} we will show is the same as A_{comp}:

- vars(A_{union}) = vars(A_1) $\cup$ vars(A_2) = $V_{P_1} \cup V_{P_2} = V_{comp}$,

- out(A_{union}) = out(A_1) $\cup$ out(A_2) = $\{\pi_w : w \in W_{P_1} \cup W_{P_2}\}$ = out(A_{comp}),

- states(A_{union}) = $\prod_{v \in V_{comp}} X_v$ = states(A_{comp}),

- start(A_{union}) = $\prod_{v \in V_{comp}} I_v$ = start(A_{comp}),

- steps(A_{union}) = $\{(s, \pi_w, s') : s'(v) = s(v)$ if $v \notin V_w$,
  $s'(v) = f^v(s)$ if $v \in V_w$, for all $w \in W_{P_1} \cup W_{P_2}$ $\}$ = steps(A_{comp}),

- $\text{part}(A_{union}) = \{\pi : \pi \in \text{acts}(A_{union})\} = \text{part}(A_{comp})$,

which implies the theorem. ■

The Union Theorem in [CM] (page 155) states the following:

1. p *unless* q in $P_1 \,[\!]\, P_2$ = p *unless* q in $P_1 \wedge$ p *unless* q in P_2.
2. p *ensures* q in $P_1 \,[\!]\, P_2$ = [p *ensures* q in $P_1 \wedge$ p *unless* q in P_2] $\vee$ [p *ensures* q in $P_2 \wedge$ p *unless* q in P_1
3. (*Fixed Point* of $P_1 \,[\!]\, P_2$) = (*Fixed Point* of P_1) $\wedge$ (*Fixed Point* of P_2)

In addition, the following corollaries are stated:

1. p is *stable* in $P_1 \,[\!]\, P_2$ = (p is *stable* in P_1) $\wedge$ (p is *stable* in P_2
2. p *unless* q in P_1 and p is *stable* in $P_2 \Rightarrow$ p *unless* q in $P_1 \,[\!]\, P_2$
3. p is *invariant* in P_1 , p is *stable* in $P_2 \Rightarrow$ p is *invariant* in $P_1 \,[\!]\, P_2$
4. p ensures q in P_1, p is *stable* in $P_2 \Rightarrow$ p ensures q in $P_1 \,[\!]\, P_2$
5. if any of the following properties holds in P_1, where p is a local predicate of P_1, then it also holds in $P_1 \,[\!]\, P_2$, for any P_2: p *unless* q, p ensures q, p is *invariant.*

All of these properties are also true for UNITY Automata:

1. P *unless* Q in $A_1 \,[\!]\, A_2$ = P *unless* Q in $A_1 \wedge$ P *unless* Q in A_2.
2. P *ensures* Q in $A_1 \,[\!]\, A_2$ = [P *ensures* Q in $A_1 \wedge$ P *unless* Q in A_2] $\vee$ [P *ensures* Q in A_2 $\wedge$ P *unless* Q in A_1
3. (*Fixed Point* of $A_1 \,[\!]\, A_2$) = (*Fixed Point* of A_1) $\wedge$ (*Fixed Point* of A_2)
4. P is *stable* in $A_1 \,[\!]\, A_2$ = (P is *stable* in A_1) $\wedge$ (P is *stable* in A_2
5. P *unless* Q in A_1 and P is *stable* in $A_2 \Rightarrow$ P *unless* Q in $A_1 \,[\!]\, A_2$
6. P is *invariant* in F , P is *stable* in $A_2 \Rightarrow$ P is *invariant* in $A_1 \,[\!]\, A_2$

7. P ensures Q in F, P is *stable* in $A_2 \Rightarrow$ P ensures Q in $A_1 \parallel A_2$

8. if any of the following properties holds in A_1, where P is a local predicate of A_1, then it also holds in $A_1 \parallel A_2$, for any A_2: P *unless* q, P ensures q, p is *invariant.*

Note that the following is also true for UNITY Automata as well as UNITY programs:

1. $P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ in $A_1 \wedge P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ in $A_2 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ in $A_1 \parallel A_2$

2. $P_1 \overset{P_2 \ldots P_{k-1}}{\longmapsto} P_k$ in $A_1 \wedge P_i unless P_{i+1}$ (for all $1 \leq i \leq k-1$) in $A_2 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ in $A_1 \parallel A_2$

   All of the above are stated and proven in Chapter 4 for the more general I/O Automata case and so are not proven here.

3.5.2 Superposition

In [CM] a structuring mechanism called superposition is introduced to structure a program as a set of "layers". Each layer implements a set of concerns. A higher layer can access lower layer but lower layers do not access higher layers. For example, an application program can be viewed as a higher layer that calls on the operating system routines. However, the operating system, the lower layer in this case, does not call on an application program. The superposition composition operator in UNITY allows a higher layer to access the variables of lower layers while a lower layer cannot access those variables of the higher layers.

Given is an underlying program, variables of which are underlying variables, we want to transform the underlying program such that all of its properties are preserved and such that the transformed program have some additional specified properties. New variables called superposed variables are introduced and the underlying program is transformed leaving the assignments to underlying variables unaffected. This transformation is done by a combination of unioning and augmenting (combining) the statements of these two components. The Superposition composition operator does not restrict which statements are unioned and which are augmented thus Superposition, unlike Union, does not uniquely determine a resultant program, but rather a set of possible resultant programs. Superposition is interesting because there are

properties shared by *all* Superpositions of an underlying program and a set of added higher level statements, thus eliminating the need to completely define the resultant transformation.

A superposition in UNITY composes a UNITY program, $P_L = \{V_L, W_L\}$, with some "higher level" assignment-statements, W_H, whose assignments may use but not modify those of the lower level program and which may introduce new variables, V_H, whose initial values are known. The transformation must follow the following two rules:

1. *Augmentation Rule:* A statement $s \in W_L$ of the underlying program may be transformed into a statement $s \parallel r$, where $r \in W_H$,

2. *Restricted Union Rule:* A statement $r \in W_H$ may be added to the underlying program, P_L.

Thus the result of a superposition of P_L is a UNITY program $P_{sup} = \{V_{sup}, W_{sup}\}$ where $V_{sup} = V_L \cup V_H$ and $W_{sup} = \{w : w \in W_L \vee w \in W_H \vee w = w_L \parallel w_H$ where $w_L \in W_L$ and $w_H \in W_H\}$

We shall now introduce an analogous composition operator for I/O automata. We define a new data structure, *higher-level-characteristic*, to describe analogous higher level actions and variables in an automaton-like form.

A higher-level-characteristic is made up of a set of higher-level variables, a set of lower level variables disjoint from the higher-level variables, an action signature, a set of states, a set of steps, and a partition on its local actions. Here we define a higher-level-characteristic, H = $(V_h, V_l, Sig(H), Start(H), States(H), Steps(H), Part(H))$ such that:

- $states(H) = \prod_{v \in V_H} X_v$
- $start(H) = \prod_{v \in V_H} \{X_v$ if $v \in V_l$, I_v if $v \in V_h \}$,
- $steps(H) = \{(s, \pi_w, s') : s'(v) = f(s)\ \forall v \in V_h,\ s'(v) = s(v)\ \forall v \in V_l \}$

We define V_H to be $V_h \cup V_l$.

To superpose H on a lower level UNITY automaton A_L the following compatibility requirements must be satisfied:

- $V_l \subseteq V_L$,

- $V_h \cap V_L = \emptyset$,
- $Acts(H) \cap Acts(A_L) = \emptyset$,
- $Acts(H) = Out(H)$,
- for every state, $s \in states(H)$ there is exactly one step (s, π, s') for all $\pi \in Acts(H)$, and
- $Part(H) = \{\{\pi\} : \pi \in acts(H)\}$.

These requirements ensure that the steps in H do not change any of the variables of A_L, that A_L is not aware of the higher level variables of H, that H and A_L do not share any actions, that the actions of H are all output actions, that H is state-deterministic, and that H have singleton partition classes. The latter three requirements are to ensure that the result of the Superposition be a UNITY automaton.

A superposition of a higher-level-characteristic, H, on a UNITY Automaton A_L can be described by applying the following two rules:

1. Augmentation Rule: An action π_H in acts(H) with steps $(s_H, \pi_H, s'_H) \in steps(H)$ may be augmented on to an action π_L in acts(A_L) with steps $(s_L, \pi_L, s'_L) \in steps(A_L)$ resulting in an action π_{sup} with steps $(s, \pi_{sup}, s'), \in steps(A_{sup})$, where $s'(v) = s'_H(v), \forall v \in V_h$ for $s(v) = s_H(v)$ and $s'(v) = s'_L(v) \forall v \in V_L$ for $s(v) = s_L(v)$,
2. Restricted Union Rule: An action $\pi : \pi \in acts(H) \cup acts(A_L)$ may be added to $acts(A_{sup})$ and the following steps added to steps(A_{sup}): if $\pi \in acts(H)$:$\{(s, \pi, s') : s(v) = s_H(v) and s'(v) = s'_H(v)$ for all $v \in V_H$ and s(v) = s'(v) for all $v \notin V_H$ for all steps $(s_H, \pi, s'_H) \in steps(H)\}$, otherwise if $\pi \in acts(A_L)$: $\{(s, \pi, s') : s(v) = s_L(v) and s'(v) = s'_L(v)$ for all $v \in V_L$ and s(v) = s'(v) for all $v \notin V_L$ for all steps $(s_L, \pi, s'_L) \in steps(A_L)\}$.

The augmentation rule can be extended to enable the augmentation of more than one action of $acts(H)$ on to one action of $acts(A_L)$ or the augmentation of one action of $acts(H)$ onto more than one action of $acts(A_L)$, however, this does not add any interesting properties since the augmentation of more than one act of either component H or A_L would be equivalent to replacing those acts in that component with one augmented act.

It is not determined which acts of H will be augmented and which will be unioned. Therefore, the acts of H can be split nondeterministically into two groups: those augmented on to the acts of A_L and those unioned with the acts of A_L. We shall refer to these two groups as $acts^A(H)$ and $acts^U(H)$ respectively. The acts of A_L can be similarly split.

Let us define a mapping, M^A, from the set $acts^A(H)$ to $acts^A(A_L)$ which maps input, internal, and output actions of H to input, internal and output actions of A_L respectively. When $acts^A(H)$ are augmented onto $acts^A(A_L)$, the result is a set of acts, $acts^A(A_{sup})$.

The result of a superposition of H on A_L is an automaton, A_{sup}, with the following components:

- $V_{sup} = V_L \cup V_H$
- $acts(A_{sup}) = out(A_{sup}) = acts^A(A_{sup}) \cup acts^U(H) \cup acts^U(A_L)$
- $states(A_{sup}) = \prod_{v \in V} X_v$,
- $start(A_{sup}) = \prod_{v \in V} I_v$,
- $steps(A_{sup}) = \{(s, \pi_{sup}, s') : \pi_{sup} \in acts(A_{sup})$, where (s, π, s') is as specified by either the Augmentation Rule or the Restricted Union Rule, depending on the definition of $\pi_{sup}\}$,
- $part(A_{sup}) = \{\{\pi\} : \pi \in acts(A_{sup})\}$

By the Superposition theorem in [CM] (page 165) every property of the underlying program is a property of the transformed program. This also holds for those properties of the lower-level UNITY automata in the superposition of a UNITY automaton with a higher-level-characteristic. This shall be stated and proven in the more general I/O Automata case.

Chapter 4

Composition

In this chapter we will examine four kinds of composition of I/O Automata. In the first two sections we extend the UNITY composition operators and apply them to more general I/O Automata. In the third section we will review the usual I/O Automata definition of Composition. Finally, in the fourth section we will define a form of composition that encompasses the three previous forms of composition.

4.1 I/O Automata Union

In this section we generalize the Union composition operator for UNITY automata presented in subsection 3.5.1 to general I/O automata. Recall that a UNITY automaton is a restricted I/O automaton with only output actions, state-deterministic actions, cartesian product states, and singleton partition classes. The main differences between the Union composition operator presented in this section and that presented in for UNITY automata lie in the treatment of I/O automata that do not meet these restrictions and are therefore not UNITY automata.

I/O Automata A_1 and A_2 must satisfy the following compatibility requirements to be combined using the Union operator:

1. X_v must be the same in A_1 and A_2 for all variables v in $V_1 \cap V_2$,
2. I_v must be the same in A_1 and A_2 for all variables v in $V_1 \cap V_2$,
3. $acts(A_1) \cap acts(A_2) = \emptyset$

These compatibility requirements are the same as those for UNITY automata.

The Union of compatible I/O Automata A_1 = (vars(A_1), states(A_1), start(A_1), sig(A_1), steps(A_1), part(A_1)) and A_2 = (vars(A_2), states(A_2), start(A_2), sig(A_2), steps(A_2), and part(A_2)) yields A = (vars(A), states(A), start(A), sig(A), steps(A), and part(A)):

- vars(A) = $vars(A_1) \cup vars(A_2)$,
- states(A) = $\prod_{v \in V} X_v$
- start(A) = $\prod_{v \in V} I_v$
- out(A) = out(A_1) $\cup$ out(A_2)
- in(A) = in(A_1) $\cup$ in(A_2)
- int(A) = int(A_1) $\cup$ int(A_2)
- $part(A) = \bigcup_{i \in I} part(A_i)$, and
- $steps(A) = \{(s, \pi, s')$: there exists an i and a step $(s_i, \pi, s_i') \in steps(A_i)$ such that $s(v) = s_i(v)$ for all $v \in vars(A_i)$, $s'(v) = s_i'(v)$ for all $v \in vars(A_i)$ and $s'(v) = s(v)$ for all $v \notin vars(A_i)\}$

In subsection 3.5.1 several theorems and lemmas were stated regarding the union of two UNITY automata. All of these theorems hold for the general case also. They shall be stated and proven for SAJ-Composition in section 4.4, a generalization of union and so are not proven here.

4.2 I/O Automata Superposition

Recall that Superposition for UNITY automata involves a combination of unioning and augmenting (combining) the actions (and their steps) of a lower-level UNITY automata and a higher-level-characteristic. The Superposition composition operator does not restrict which actions are unioned and which are augmented thus Superposition, unlike Union, does not uniquely determine a resultant UNITY automaton, but rather a set of possible resultant UNITY automata.

In this section we generalize the Superposition composition operator for UNITY automata presented in subsection 3.5.2 to general I/O automata. As with the generalization of the Union composition operator, the main differences between the Superposition composition operator presented in this section and the one presented for UNITY automata lie in the treatment of I/O automata that are not UNITY automata.

Recall from subsection 3.5.2 that a higher-level-characteristic is made up of a set of higher-level variables, a set of lower level variables disjoint from the higher-level variables, an action signature, a set of states, a set of steps, and a partition of its local actions. Here we refer to a higher-level-characteristic, H = $(V_h, V_l, Sig(H), Start(H), States(H), Steps(H), Part(H))$. We define V_H to be $V_h \cup V_l$.

To superpose H on a lower level I/O automaton A_L the following compatibility requirements must be satisfied:

- $V_l \subseteq V_L$,
- $V_h \cap V_L = \emptyset$, and
- $Acts(H) \cap Acts(A_L) = \emptyset$.

These requirements ensure that the steps in H do not change any of the variables of A_L, that A_L is not aware of the higher level variables of H, and that they share no actions.

A superposition of a higher-level-characteristic, H, on a UNITY Automaton A_L can be described by applying the following two rules:

1. Augmentation Rule: An action π_H in acts(H) with steps $(s_H, \pi_H, s'_H) \in steps(H)$ may be augmented on to an action π_L in acts(A_L) with steps $(s_L, \pi_L, s'_L) \in steps(A_L)$ resulting in an action π_{sup} with steps $(s, \pi_{sup}, s'), \in steps(A_{sup})$, where $s'(v) = s'_H(v), \forall v \in V_h$ for $s(v) = s_H(v)$ and $s'(v) = s'_L(v) \forall v \in V_L$ for $s(v) = s_L(v)$, *provided* that π_H and π_L are *compatible*,

2. Restricted Union Rule: An action $\pi : \pi \in acts(H) \cup acts(A_L)$ may be added to $acts(A_{sup})$ and the following steps added to steps(A_{sup}): if $\pi \in acts(H)$:$\{(s, \pi, s') : s(v) = s_H(v) and s'(v) = s'_H(v)$ for all $v \in V_H$ and s(v) = s'(v) for all $v \notin V_H$ for all steps $(s_H, \pi, s'_H) \in steps(H)\}$,

otherwise if $\pi \in acts(A_L)$: $\{(s, \pi, s') : s(v) = s_L(v) and s'(v) = s'_L(v)$ for all $v \in V_L$ and s(v) = s'(v) for all $v \notin V_L$ for all steps $(s_L, \pi, s'_L) \in steps(A_L)\}$.

Here the augmentation rule differs from that for UNITY automata in that there is a compatibility requirement. Compatibility here is defined as being enabled for all of the same states. Formally, π_H and π_L are compatible if for any step $(s_L, \pi_L, s'_L) \in steps(A_L)$ there is a step $(s_H, \pi_H, s'_H) \in steps(H)$ where $s_H(v) = s_L(v)$ for all $v \in V_l$, and for any step $(s_H, \pi_H, s'_H) \in steps(A_H)$ there is a step $(s_L, \pi_L, s'_L) \in steps(L)$ where $s_H(v) = s_L(v)$ for all $v \in V_l$.

The augmentation rule can be extended to enable the augmentation of more than one action of $acts(H)$ on to one action of $acts(A_L)$ or the augmentation of one action of $acts(H)$ onto more than one action of $acts(A_L)$, however, this does not add any interesting properties since the augmentation of more than one action of either component, H or A_L, would be equivalent to replacing those actions in that component with one augmented act.

It is not determined which acts of H will be augmented and which will be unioned. Therefore, the acts of H can be split nondeterministically into two groups: those augmented on to the acts of A_L and those unioned with the acts of A_L. We shall refer to these two groups as $acts^A(H)$ and $acts^U(H)$ respectively. The acts of A_L can be similarly split.

Let us define a mapping, M^A, from the set $acts^A(H)$ to $acts^A(A_L)$. When $acts^A(H)$ are augmented onto $acts^A(A_L)$, the result is a set of acts, $acts^A(A_{sup})$. We will refer to the act resulting from an augmentation of an act $\pi_H \in acts^A(H)$ onto an act $\pi_L \in acts^A(A_L)$ as $aug(\pi_L)$ or $aug(\pi_H)$.

The result of a superposition of H on A_L is an automaton, A_{sup}, with the following components:

- $V_{sup} = V_L \cup V_H$
- $acts(A_{sup}) = acts^A(A_{sup}) \cup acts^U(H) \cup acts^U(A_L)$
- $states(A_{sup}) = \prod_{v \in V} X_v$,
- $start(A_{sup}) = \prod_{v \in V} I_v$,

- $steps(A_{sup}) = \{(s, \pi_{sup}, s') : \pi_{sup} \in acts(A_{sup})$, where (s, π, s') is as specified by either the Augmentation Rule or the Restricted Union Rule, depending on the definition of $\pi_{sup}\}$,
- $part(A_{sup}) = \{aug(C) : C \in part(A_L) \cup part(H)\}$

Here the augmentation of a class C, $aug(C)$, denotes a class equal to C with those actions π in $acts^A(A_L) \cup acts^A(H)$ replaced by $aug(\pi)$. Note that here $part(A_{sup})$ is no longer a true partition in that there are actions that are in more than one class. This is necessary when combining actions, however, because actions of possibly more than one component are being combined. An alternative to allowing an action to be in more than one class is to combine these classes and thus combine the components. This is not desirable because then the actions of one of the original components may be ignored always according to the fairness rule since the actions of the combined class disjoint from the actions of the original component may always be selected instead and still satisfy the fairness condition.

By the Superposition theorem in [CM] (page 165) every property of the underlying program is a property of the transformed program. This also holds for those properties of the lower-level I/O automata in the superposition of a I/O automaton with a higher-level-characteristic.

Theorem 38 *If any of the following properties (or their analogous local properties) is a property of the lower-level I/O Automaton, it is a property of the transformed I/O Automaton:* unless, ensures, $\mapsto$, detects, until, stable, invariant

Proof: Since none of the variables of the lower-level I/O Automaton can be modified by any of the actions of the higher-level-characteristic by definition, there is no way that a property of the lower-level I/O Automaton could not be a property of the transformed I/O Automaton. ■

One notable exception to the above theorem is *Fixed Point.* This is due to the fact that *Fixed Point* involves a particular state in the lower-level I/O Automaton. There may be some action in the higher-level-characteristic that causes the state of one of its higher level variables to oscillate. If the transposed I/O Automaton contains this actions as a result of a restricted union, then the transposed I/O Automaton would not have a *Fixed Point*, even though with respect to the state of the lower-level variables, there is an effective *Fixed Point.* For this reason we defined a new kind of "Fixed Point" with respect to a subset of the I/O Automaton

variables, the *variable set fixed point.* This is useful in reasoning about a set of states where states where certain variables have been fixed (for example, to detect a terminating condition) in addition to reasoning about composed automata.

Theorem 39 *If the lower-level I/O Automaton has a* fixed point *or a* local fixed point, *the transformed I/O Automaton will have a* variable set fixed point *or a* variable set local fixed point *respectively in the variable set* $V_{lower-level}$.

Proof: Since none of the variables of the lower-level I/O Automaton can be modified by any of the actions of the higher-level-characteristic by definition, if the lower-level I/O Automaton has a *fixed point* or *local fixed point*, the transformed I/O Automaton will have a *variable set fixed point* or *variable set local fixed point* respectively in the variable set of the lower-level I/O Automaton. ■

Theorem 40 *If any of the following properties (or their analogous local properties) is a property solely of the variables initialized by the higher-level-characteristic, it is a property of the transformed I/O Automaton:* unless, ensures, $\mapsto$, detects, until, stable, invariant

Proof: Since none of the variables initialized by the higher-level-characteristic can be modified by any of the actions of the lower-level I/O Automaton by definition, there is no way that a property of those variables in the higher-level-characteristic could not be a property of the transformed I/O Automaton. ■

Theorem 41 *If the higher-level-characteristic has a* fixed point *or a* local fixed point, *the transformed I/O Automaton will have a* variable set fixed point *or a* variable set local fixed point *respectively in the variable set* $V_{higher-level-characteristic}$.

Proof: Since none of the variables initialized by the higher-level-characteristic can be modified by any of the actions of the lower-level I/O Automaton by definition, if the higher-level-characteristic has a *fixed point* or *local fixed point*, the transformed I/O Automaton will have a *variable set fixed point* or *variable set local fixed point* respectively in the variable set initialized by the higher-level-characteristic. ■

4.2.1 Example : Global Snapshot

In [CM] the problem of recording global states of a program is solved using superposition. The original program is treated as the lower-level program and the higher-level statements do the recording of the states. A similar approach may be used to find an I/O Automata that solves this problem.

Here we propose a higher-level-characteristic for recording the "state" (i.e. the values of the variables) of a lower-level I/O Automata:

- $V_h = \{v.record, v.recorded : v \in V_l\}$
- $I_h = \{v.recorded = false\}$
- $start(H) = \prod_{v \in V_h} nil$
- $states(H) = \prod_{v \in V_H} X_v$
- $acts(H) = out(H) = \{\pi_{rec}\}$
- $steps(H) = \{(s, \pi_{rec}, s') :$ for all $v \in V_l$ $s'(v.record) = s(v), s'(v.recorded) = true$ if $s(v.recorded) = false\}$

This solution allows for the simultaneous recording of all the variables in the underlying program. However, while this solution is suitable for sequential machines or parallel synchronous machines, it is not suitable for distributed machines.

We may refine our higher-level-characteristic to allow the state to be recorded a bit at a time. In this case we want the present record of the state to be part of a possible reachable state from the initial conditions of the lower level automaton. Furthermore we want it also to be part of a state from which the actual present state of the automaton is reachable.

Let us consider the example of a distributed system of processors sending messages along channels. We may formulate a higher-level-characteristic to record the global state based on the solution in [CL] which uses the sending of markers along channels to determine when to record the state of a process and when and what to record for the state of the channel.

It is assumed here that the underlying I/O Automaton A is actually distributed which is defined as having processors and channels and the following communication actions, $comm(A) \subseteq acts(A)$:

1. send-message($p_i, c_{i,j}, msg$)

   and

2. receive-message($c_{j,i}, p_i, msg$)

A higher-level-characteristic, H, that follows the approach specified in [CL] can be superposed onto such an I/O Automaton, A. One such higher-level-characteristic may be defined as follows:

- $V_l = \{v : v \in vars(A)\}$
- $V_h = \{v.record, v.recorded, v.tracking : v \in vars(A)\}$
- $I_h = \{v.recorded = false\}$
- $states(H) = \prod_{v \in V_h \cup V_l} X_v$
- $acts(H) = in(H) \cup out(H) \cup int(H)$
- $in(H) = \{record - state\} \cup comm(A)$
- $out(H) = \{done\}$
- $int(H) = \emptyset$
- $steps(H) =$
  $\{(s, record{-}state, s') : s'(p_i.record) = s(p_i), s'(p_i.recorded) = true$, for some i, $s'c_{i,j}.tracking = s(c_{i,j}||marker)$ for all j, $s'c_{j,i}.tracking = empty$ for all j, $s'c_{j,i}.record = empty$ for all j, [enabled] if $s(v.recorded) = false$ for all $v \in V_l\}$

  $record - state$ indicates the beginning of the recording process. One of the processor's present state is recorded and its outgoing channels begin to be "tracked" with the beginning of the "tracking" being indicated by a marker and its incoming channels begin to be tracked also.

  $\cup \{(s, send{-}message(p_i, c_{i,j}, msg) \in comm(A), s') : s'(c_{i,j}.tracking) = s(c_{i,j}.tracking||msg)$

}

When a message is sent by a processor to another the tracking of the channel between them continues to be updated.

$\cup$ $\{(s, receive - message(c_{j,i}, p_i, msg), s') : s'(c_{j,i}.tracking) = tail(s(c_{j,i}.tracking))$, [enabled] if $p_i.recorded = false \wedge head(s(c_{j,i}.tracking)) \neq marker\}$

When a message is received by an unrecorded processor in A the tracking of the incoming channel continues to be updated as long as the first element of the tracking is not a marker.

$\cup$ $\{(s, receive-message(c_{j,i}, p_i, msg), s') : s'(c_{i,j}.tracking) = tail(s(c_{i,j}.tracking)), s'(c_{i,j}.record) = s(c_{i,j}.record)||head(s(c_{i,j}.tracking))$, [enabled] if $p_i.recorded = true \wedge head(s(c_{j,i}.tracking)) \neq marker\}$

When a message is received by a recorded processor in A the tracking and record of the incoming channel continues to be updated as long as the first element of the tracking is not a marker. The record of the incoming channel is the sequence of messages received since the recording of the recorded processor. Note, however that the incoming channel is not recorded until a marker heads its tracking.

$\cup$ $\{(s, receive - message(c_{j,i}, p_i, msg), s') : s'(c_{j,i}.record) = empty,\ s'(p_i.record) = s(p_i),$ $s'(c_{j,i}.recorded) = true,\ s'(p_i.recorded) = true,\ c_{j,i}.tracking = empty,\ c_{j,i}.record = empty$, [enabled] if $p_i.recorded = false \wedge head(s(c_{j,i}.tracking)) = marker\}$
If the first element of the tracking of the incoming channel to an unrecorded processor is a marker, then the state of the unrecorded processor is recorded to be its state before the action and the state of the channel to it is recorded to be empty.

$\cup$ $\{(s, receive-message(c_{j,i}, p_i, msg), s') : s'(c_{i,j}.tracking) = tail(s(c_{i,j}.tracking)), s'(c_{i,j}.recorded)$ $true$, [enabled] if $p_i.recorded = true \wedge head(s(c_{j,i}.tracking)) \neq marker\}$

If the first element of the tracking of the incoming channel to a recorded processor is a marker then the record of the incoming channel has the sequence of messages received from the recording of the processor until the receiving of the marker and thus is complete at which time the incoming channel is marked as recorded.

$\cup \{(s, done, s') : \text{[enabled] if } v.recorded = true \text{ for all } p_i, c_{i,j} \in vars(A)\}$

The recording process is finished (outputs "done"), when all the processors and channels are recorded.

Here p_i represents a processor i and $c_{i,j}$ represents a channel from processor i to processor j.

4.3 I/O Automata Composition

This is the composition operator defined in [LT] and described briefly in the introduction. We shall represent this I/O Automata composition by the symbol $\oplus$. The I/O Automata composition operator may only compose strongly compatible I/O Automata, A_i with compatibility defined as:

1. $out(A_i) \cap out(A_j) = \emptyset$,
2. $int(A_i) \cap acts(A_j) = \emptyset$, and
3. no action is shared by infinitely many A_i

The definition of the I/O Automata composition of compatible I/O Automata A_i ($i \in I$ of finite size) to yield A is defined in [LM] as follows:

1. $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$,
2. $out(A) = \bigcup_{i \in I} out(A_i)$,
3. $int(a) = \bigcup_{i \in I} int(A_i)$,
4. $states(A) = \prod_{i \in I} states(A_i)$,

5. $start(A) = \prod_{i \in I} start(A_i)$,

6. $part(A) = \bigcup_{i \in I} part(A_i)$, and

7. $steps(A) = \{(s_i, \pi, s_i') :$ for all $i \in I$, if $\pi \in acts(A_i)$ then $(s_i, \pi, s_i') \in steps(A_i)$, otherwise $s_i = s_i'$.

This composition operator may be seen as a generalization of the union operator described in section 4.1 if the I/O automata composed have disjoint variables. It is a generalization in that it allows input and output actions and shared actions (with some restrictions). However, it yields different results than the union composition operator when there the variables of the components are not disjoint because of the definition of $states(A)$ of the resultant I/O automata. Specifically, if A_1 and A_2 do not have disjoint variables, there will be states in $A_1 \oplus A_2$ that are not self-consistent because the cartesian product of the states in A_1 and A_2 have a shared variable component. This observation indicates that union may be better suited for reasoning about shared-variable systems and this composition operator may be better suited for reasoning about distributed systems.

4.4 I/O Automata SAJ-Composition

The definition of I/O automata composition of the previous section has two drawbacks. The first is that it does not allow outputs to be shared by components. This may be a desirable property. For example if we have a bank account accessible from several ATMs. To use the composition operator defined in the previous section, each output action of the ATM automaton accessing the account (the account input actions) would have to be distinct to each ATM. A more realistic model would be one where each ATM would have a shared output action that accesses the account. The second drawback is when trying to represent a shared variable. If for some shared action step one automaton increments this variable and the other decrements it, in the definition of composition in [LT] described in the previous section, that would be allowed and this contradictory state may be reached since the value of the shared variable would be recorded in the state twice rather than once. This form of composition guarantees that this kind of inconsistency will never occur by requiring that all shared action that may be executed by two components simultaneously change all shared variables in the same way.

We shall represent the Shared Action Join Composition operator by the symbol $\bowtie$.

Let us define SAJ-Composition Compatible I/O Automata. Two I/O Automata, A_1 and A_2 are compatible if:

1. X_v must be the same in A_1 and A_2 for all variables v in $V_1 \cap V_2$,
2. I_v must be the same in A_1 and A_2 for all variables v in $V_1 \cap V_2$,
3. $int(A_i) \cap acts(A_j) = \emptyset$, and
4. no action is shared by infinitely many A_i
5. for all $\pi \in acts(A_1) \cap acts(A_2)$ for every step (s_i, π, s_i') in A_i there must exist a step (s_j, π, s_j') in A_j, where $s_i(v) = s_j(v)$ and $s_i'(v) = s_j'(v)$ for all $v \in vars(A_1) \cap vars(A_2)$

The last compatibility condition states that shared actions must be enabled the same way and act the same way on their common variables in their corresponding states in both automata.

The SAJ-Composition of compatible I/O Automata A_1 = (vars(A_1), states(A_1), start(A_1), sig(A_1), steps(A_1), part(A_1)) and A_2 = (vars(A_2), states(A_2), start(A_2), sig(A_2), steps(A_2), and part(A_2)) yields A = (vars(A), states(A), start(A), sig(A), steps(A), and part(A)):

- $vars(A) = vars(A_1) \cup vars(A_2)$,
- $states(A) = \prod_{v \in V} X_v$
- $start(A) = \prod_{v \in V} I_v$
- $out(A) = out(A_1) \cup out(A_2)$
- $in(A) = in(A_1) \cup in(A_2)$ - $out(A)$
- $int(A) = int(A_1) \cup int(A_2)$
- $steps(A) = \{(s, \pi, s')$: for every $(s_i, \pi, s_i') \in steps(A_i)$ (for every A_i) where $s(v) = s_i(v)$ for all $v \in vars(A_i)$ then $s'(v) = s_i'(v)$ for all $v \in vars(A_i)$ and $s'(v) = s(v)$ for all $v \notin vars(A_i)\}$
- $part(A) = \prod_{i \in I} part(A_i)$

Here again, like in Superposition, we may have a shared action in two different classes in the partition of A. This is not a problem since we require that the component actions of this shared action be enabled at the same time. Therefore the resulting composed automaton will have fair executions that are the interleavings of fair executions of its components. Actions are combined here as in the augmentation step of Superposition, however the steps combined are not nondeterministically chosen but rather are those actions shared.

Theorem 42 *If A_1 and A_2 have no variables in common, $A_1 \oplus A_2 = A_1 \bowtie A_2$.*

Proof: Obvious. By definition. ■

Theorem 43 *If A_1 and A_2 have no actions in common, $A_1 \,[\!]\, A_2 = A_1 \bowtie A_2$.*

Proof: Obvious. By definition. ■

The following are the proofs of the theorems and lemmas stated in subsection 3.5.1:

Lemma 44 *P* unless *Q in $A_1 \bowtie A_2 \Leftrightarrow P$* unless *$Q$ in $A_1 \wedge P$* unless *Q in A_2.*

Proof: By the definition of *unless*, if *P unless Q* in $A_1 \bowtie A_2$ then for all $\pi \in acts(A_1 \bowtie A_2)$, $\{P \cap \sim Q\}\pi\{P \cup Q\}$. Since $acts(A_1 \bowtie A_2) = \bigcup\, acts(A_i)$, for all $\pi \in acts(A_1 \bowtie A_2)$, $\{P \cap \sim Q\}\pi\{P \cup Q\}$ iff for all $\pi \in acts(A_i)$, $\{P \cap \sim Q\}\pi\{P \cup Q\}$. So $A_1 \bowtie A_2$ iff A_i satisfies *P unless Q* for all i. ■

Corollary 45 *P is* stable *in $A_1 \bowtie A_2 \Leftrightarrow$ (P is* stable *in A_1) $\wedge$ (P is* stable *in A_2).*

Proof: This follows from the definition of *stable* and Lemma 44. ■

Corollary 46 *P* unless *Q in A_1 and P is* stable *in $A_2 \Rightarrow P$* unless *Q in $A_1 \bowtie A_2$.*

Proof: By the definition of *stable*, if A_2 satisfies P is *stable* then it trivially satisfies *P unless Q* for any Q. Thus by Lemma 44, this lemma holds. ■

Corollary 47 *P is* invariant *in A_1 and P is* stable *in $A_2 \Rightarrow P$ is* invariant *in $A_1 \bowtie A_2$.*

Proof: By the compatibility requirement of SAJ-composition, if $start(A_1) \subseteq P$ then $start(A_2) \subseteq P$. Therefore, if P is *invariant* in A_1 and P is *stable* in A_2, then P must be *invariant* in A_2 also. By the definition of $start(A_1 \bowtie A_2)$, $start(A_1 \bowtie A_2) \subseteq P$. Thus by the definition of *stable* and by Lemma 44, this lemma holds. ■

Lemma 48 *P* ensures *Q in* $A_1 \bowtie A_2 \Leftrightarrow$ *[P* ensures *Q in* $A_1 \wedge$ *P* unless *Q in* A_2*] ∨ [P* ensures *Q in* $A_2 \wedge$ *P* unless *Q in* A_1*].*

Proof: Since $acts(A_1 \bowtie A_2) = \bigcup_{i \in I} acts(A_i)$ and $part(A_1 \bowtie A_2) = \bigcup_{i \in I} part(A_i)$ then there exists a class $C \in part(A_1 \bowtie A_2)$ such that $\{P \cap \sim Q\}\pi\{Q\}$ for all $\pi \in C \Leftrightarrow$ for some $i \in I$ there must exist a class $C \in part(A_i)$ such that $\{P \cap \sim Q\}\pi\{Q\}$ for all $\pi \in C$. By Lemma 44 we know that P *unless* Q in $A_1 \bowtie A_2$ and there exists a class $C \in part(A_1 \bowtie A_2)$ such that $\{P \cap \sim Q\}\pi\{Q\}$ for all $\pi \in C \Leftrightarrow$ P *unless* Q in $A_1 \wedge$ P *unless* Q in A_2. Combining these we have P *unless* Q in $A_1 \bowtie A_2 \Leftrightarrow$ P *unless* Q in $A_1 \wedge$ P *unless* Q in A_2 and for some $i \in I$ there must exist a class $C \in part(A_i)$ such that $\{P \cap \sim Q\}\pi\{Q\}$ for all $\pi \in C$. By the definition of *ensures*, P *ensures* Q in $A_1 \bowtie A_2$, $\Leftrightarrow$ *P unless Q* in $A_1 \bowtie A_2$ and there exists a class $C \in part(A_1 \bowtie A_2)$ such that $\{P \cap \sim Q\}\pi\{Q\}$ for all $\pi \in C$. Therefore, P *ensures* Q in $A_1 \bowtie A_2 \Leftrightarrow$ [P *ensures* Q in $A_1 \wedge$ P *unless* Q in A_2] ∨ [P *ensures* Q in $A_2 \wedge$ P *unless* Q in A_1]. ■

Corollary 49 *P ensures Q in* A_1 *and P is* stable *in* $A_2 \Rightarrow$ *p ensures Q in* $A_1 \bowtie A_2$

Proof: This holds by the definition of *stable* and Lemma 48. ■

Corollary 50 *If any of the following properties holds in* A_1*, where P is a predicate dependent only on the variables of* A_1 *disjoint from those of* A_2*, then it also holds in* $A_1 \bowtie A_2$*, for any* A_2*: P* unless *Q*, *P* ensures *Q*, *P is* invariant.

Proof: If *P* is a predicate dependent only on the (disjoint) variables of A_1 and *P unless Q*, *P ensures Q*, or *P* is *invariant* in A_1, then it trivially holds for A_2 and then by either Lemma 44 or Lemma 48, this lemma holds. ■

Lemma 51 *(State s is a* Fixed Point *of* $A_1 \bowtie A_2$*)* $\Leftrightarrow$ *(*s_1 *is a* Fixed Point *of* A_1*)* $\wedge$ *(*s_2 *is a* Fixed Point *of* A_2*) where* $s_1(v) = s(v)$ *for all* $v \in vars(A_1)$ *and* $s_2(v) = s(v)$ *for all* $v \in vars(A_2)$.

Proof: By the definition of *Fixed Point*, state *s* is a *Fixed Point* in $A_1 \bowtie A_2$ iff for all $(s, \pi, s') \in steps(A_1 \bowtie A_2)$, $s = s'$ if $\pi \in acts(A_1 \bowtie A_2)$. By the definition of $(s, \pi, s') \in steps(A_1 \bowtie A_2)$, $s = s'$ iff for all $i \in I$ and for all $\pi \in acts(A_i)$, $(s_i, \pi, s_i') \in steps(A_i) \Rightarrow s_i = s_i'$ where for all $v \in vars(A_i)$, $s(v) = s_i(v)$. Thus for all $i \in I$, *s* is a *Fixed Point* for all A_i ■

Note that although a limitation of composition operators is that a property of the form $P \mapsto Q$ cannot be asserted in $A_1 \bowtie A_2$ even though it holds in both A_1 and A_2, the following is true for I/O Automata:

Lemma 52 $P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \wedge P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_2 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 48. ■

Lemma 53 $P_1 \overset{P_2 \ldots P_{k-1}}{\longmapsto} P_k$ *in* $A_1 \wedge P_i$ unless P_{i+1} *(for all* $1 \leq i \leq k-1$*) in* $A_2 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 48. ■

Lemma 54 *[[*P_m ensures P_{m+1} *in* A_i*] for some* $i \in I \wedge$ *[*P_m unless P_{m+1} *in* A_j*] for all* $j \in I$*]] for all* $1 \leq i \leq k-1 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 48. ■

These are the analogous local lemmas:

Lemma 55 P local-unless Q *in* $A_1 \bowtie A_2 \Leftrightarrow P$ local-unless Q *in* $A_1 \wedge P$ local-unless Q *in* A_2.

Proof: Proof is analogous to that of Lemma 44. ■

Corollary 56 P *is* locally stable *in* $A_1 \bowtie A_2 \Leftrightarrow$ *(*P *is* locally stable *in* A_1*)* $\wedge$ *(*P *is* locally stable *in* A_2

Proof: This follows from the definition of *locally stable* and Lemma 55. ■

Corollary 57 P local-unless Q *in* A_1 *and* P *is* locally stable *in* $A_2 \Rightarrow P$ local-unless Q *in* $A_1 \bowtie A_2$

Proof: By the definition of *locally stable*, if A_2 satisfies P is *locally stable* then it trivially satisfies P *local-unless* Q for any Q. Thus by Lemma 55, this lemma holds. ■

Corollary 58 P *is* local invariant *in* A_1 *and* P *is* locally stable *in* $A_2 \Rightarrow P$ *is* local invariant *in* $A_1 \bowtie A_2$

Proof: By the compatibility requirement of SAJ-composition, if $start(A_1) \subseteq P$ then $start(A_2) \subseteq P$. Therefore, if P is *local invariant* in A_1 and P is *locally stable* in A_2, then P must be *local invariant* in A_2 also. By the definition of $start(A_1 \bowtie A_2)$, $start(A_1 \bowtie A_2) \subseteq P$. Thus by the definition of *locally stable* and by Lemma 55, this lemma holds. ■

Lemma 59 *P* local-ensures *Q in $A_1 \bowtie A_2 \Leftrightarrow$ [P* local-ensures *Q in $A_1 \wedge P$* local-unless *Q in A_2] $\vee$ [P* local-ensures *Q in $A_2 \wedge P$* local-unless *Q in A_1].*

Proof: Proof is analogous to that of Lemma 48. ■

Corollary 60 *P local-ensures Q in A_1 and P is* locally stable *in $A_2 \Rightarrow p$ local-ensures Q in $A_1 \bowtie A_2$*

Proof: This holds by the definition of *locally stable* and Lemma 59. ■

Corollary 61 *If any of the following properties holds in A_1, where P is a predicate dependent only on the (disjoint) variables of A_1, then it also holds in $A_1 \bowtie A_2$, for any A_2: P* local-unless *Q, P* local-ensures *Q, P is* local invariant.

Proof: If P is a predicate dependent only on the (disjoint) variables of A_1 and P *local-unless* Q, P *local-ensures* Q, or P is *local invariant* in A_1, then it trivially holds for A_2 and then by either Lemma 55 or Lemma 59, this lemma holds. ■

Lemma 62 *(*Local Fixed Point *of $A_1 \bowtie A_2$) $\Leftrightarrow$ (*Local Fixed Point *of A_1) $\wedge$ (*Local Fixed Point *of A_2)*

Proof: By the definition of *Local Fixed Point*, state s is a *Local Fixed Point* in $A_1 \bowtie A_2$ iff for all $(s, \pi, s') \in steps(A_1 \bowtie A_2)$, $s = s'$ if $\pi \in local(A_1 \bowtie A_2)$. By the definition of $(s, \pi, s') \in steps(A_1 \bowtie A_2)$, $s = s'$ iff for all $i \in I$ and for all $\pi \in local(A_i)$, $(s_i, \pi, s_i') \in steps(A_i)$ $\Rightarrow s_i = s_i'$ where for all $v \in vars(A_i)$, $s(v) = s_i(v)$. Thus for all $i \in I$, s is a *Local Fixed Point* for all A_i ■

Lemma 63 $P \overset{\ell P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \wedge P \overset{\ell P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_2 \Rightarrow P \overset{\ell P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 59. ■

Lemma 64 $P_1 \overset{\ell P_2 \ldots P_{k-1}}{\longmapsto} P_k$ *in* $A_1 \wedge P_i \text{local} - \text{unless} P_{i+1}$ *(for all* $1 \leq i \leq k-1$*) in* $A_2 \Rightarrow P \overset{\ell P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 59. ■

Lemma 65 *[[*$P_m \text{local} - \text{ensures} P_{m+1}$ *in* A_i*] for some* $i \in I \wedge$ *[*$P_m \text{local} - \text{unless} P_{m+1}$ *in* A_j*] for all* $j \in I$*]] for all* $1 \leq i \leq k-1 \Rightarrow P \overset{P_2 \ldots P_{k-1}}{\longmapsto} Q$ *in* $A_1 \bowtie A_2$

Proof: This holds by induction on P_i using Lemma 59. ■

Chapter 5

Randomized Algorithms

For randomized algorithms there are times we would like to represent a random choice in the component of the I/O automaton chosen. This case could be modeled by associating a probability with each class in $part(A)$ thus making executions of A fair with probability one. More often, we would like to represent a random choice of action in some component. In a possible modeling of this case it is the actions within the class that are associated with a probability assignment. Here such a modification to the I/O Automaton model is presented and analyzed.

5.1 Randomized I/O Automata

We augment the I/O Automaton model to allow discussion of algorithms in which there is a random choice of variable value or chosen action with the probability mass or density function known. The probability mass function represents the probability function of a choice of a discrete random variable or randomly choosing a transition out of a countably infinite number of possibilities. The probability density function represents the probability function of a choice of a continuous random variable or randomly choosing a transition out of an uncountably infinite number of possibilities. This randomized I/O automaton is an I/O automaton with an additional component, $prob(A)$ which associates a probability mass or density function on the set of steps of enabled actions in every class in $part(A)$ to every state in $states(A)$. In other words for every state $s \in states(A)$ and every class $C \in part(A)$ there is an associated

probability mass or density function on the steps $(s, \pi \in C, s') \in steps(A)$ thus the probabilities of steps with actions from a class in $part(A)$ originating from a state in $states(A)$ sum to one.

More formally, a randomized I/O automaton is an I/O automaton with the additional component $prob(A) = \{prob(s, C) : s \in states(A), C \in part(A)\}$ where

- $prob(s, C) = \{P^s_C(\pi, s') : s, s' \in states(A), C \in part(A), \pi \in C, \}$ where P^s_C is a probability mass function and $P^s_C(\pi, s')$ represents the probability of the step $(s, \pi, s') \in steps(A)$ being executed if component C has been chosen.

Note that $\sum_{\pi \in C, (s,\pi,s') \in steps(A)} P^s_C(\pi, s') = 1$. This definition can be extended for the continuous case using a probability density function, however we will concentrate on the discrete case.

We shall use the following notation: $p(s, \pi, s') = P^s_{C_\pi}(\pi, s')$. Additionally, it is often easier to state the association of the probability of a given step in $steps(A)$. Thus for such cases, a step of A will be represented as $((s, \pi, s'), p)$ where $p = p(s, \pi, s')$ or as (s, π, s') where p is implied to be 1.

When composing randomized I/O automata (in any of the composition methods described in this paper), the probabilities of the steps of the composition is the same as the probability of the analogous step in the component. There will be no conflict due to shared actions for the Union composition operator or the Composition as defined in [LT] with this definition of the probability since shared actions are local to at most one component for these composition methods and thus only one component may specify a probability. For Superposition and SAJ-composition where a local action may be an action of more than one component, it may indeed have more than one probability assigned to it. This is not a problem since a step of such an action would have a different probability depending on which component executes it and can almost be considered to be two different actions with the same effect.

5.2 UNITY Proofs Concepts for Randomized I/O Automata

Most randomized algorithms satisfy all the necessary safety properties but only satisfy the progress properties with probability one. Therefore in this section we expand UNITY progress proof concepts to pertain to randomized I/O automata. We shall concentrate on the discrete

case in this paper. The ideas presented can be generalized to apply to the continuous case as well.

A randomized I/O automaton A satisfies P *ensures* Q *with probability one* iff A satisfies P *unless* Q and there exists a class C such that for every state s in $P \cap \sim Q$, there exists a step $(s, \pi, s') \in steps(A)$ where $\pi \in C$, $s' \in Q$ and $p \geq \epsilon > 0$. Note that if A is a randomized I/O automaton that satisfies P *ensures* Q then it also satisfies P *ensures* Q *with probability one* by definition of the latter.

A satisfies $P \mapsto Q$ *with probability one* iff there exists a sequence $P_1, \ldots P_k$, $(k \geq 2)$ of sets where $P_1 = P$, $P_k = Q$, and A satisfies P_i *ensures* P_{i+1} *with probability one* for all $1 \leq i \leq k-1$.

The analogous local properties can be similarly defined.

Lemma 66 *If A satisfies* P ensures Q with probability one, *then in any fair execution $\alpha = s_0 \pi_1 s_1 \pi_2 \ldots$ of A, if $s_i \in P$ then with probability one there is an index $k \geq i$ such that $s_k \in Q$ and $\forall j, (i \leq j < k), s_j \in P \cap \sim Q$.*

Proof: If s_i is also in Q then $i = k$ and the lemma is true, otherwise if s_i is in P but not in Q, then since A satisfies P *unless* Q, by Lemma 1 we know that if there is a state, $s_k \in Q, (k \geq i)$ where $s_j \notin Q$ for all $k > j \geq i$ (i.e. s_k is the first state in Q after s_i), then for all $k > j \geq i$, $s_j \in P \cap \sim Q$. Now we must show that such a state exists with probability one. We know that there exists an class C of local actions of A such that for all states in $P \cap \sim Q$ there exists some action in C with a nonzero probability that will yield a new state that satisfies Q. We also know that in every state in $P \cap \sim Q$ there is such an action.

Case 1: α is finite. The fairness condition states that if α is a finite execution, then no action of C is enabled from the final state of α. Therefore α may not end in a state in $P \cap \sim Q$. Thus by Lemma 1 we know that since s_j cannot be in $P \cap \sim Q$ for all $j \geq i$ then there must exist some $k \geq i$ such that $s_k \in Q$ and for all $i \leq j < k$, $s_j \in P \cap \sim Q$ and the lemma holds.

Case 2: α is infinite. The fairness condition states that if α is an infinite execution, then either actions from C must appear infinitely often in α, or states from which no action of C is enabled appear infinitely often in α. Since in every state in $P \cap \sim Q$ there is an action in C enabled, we know that the latter cannot be the case if states remain in $P \cap \sim Q$. Therefore actions from C appear infinitely often in α. By the definition of *ensures with probability one* we

know that there is a step from every state in $P \cap \sim Q$ with nonzero probability that results in a state in Q. Therefore from the definition of a randomized I/O automaton we know that the probability of choosing an action of C that results in a state in $P \cap \sim Q$ is less than one. Let $p^{\max}_{P \cap \sim Q \rightarrow P \cap \sim Q} = \max\{p(s, \pi, s') : s, s' \in P \cap \sim Q, \pi \in C\}$ representing the maximum probability of choosing an action of C that results in a state in $P \cap \sim Q$. Thus, in order for the states in α to remain in $P \cap \sim Q$ such actions must be chosen an infinite number of times. The probability of such an execution is then less than or equal to $(p^{\max}_{P \cap \sim Q \rightarrow P \cap \sim Q})^n$ where n is the number of times such an action is chosen. Since we know n is infinite, then this probability is zero. Thus with probability one an action in C is chosen that results in a state in Q and the lemma holds. ■

Lemma 67 *If A satisfies* $P \mapsto Q$ with probability one *then in a fair execution* $\alpha = s_0 \pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then with probability one there exists an* $m \geq i$, *such that* $s_m \in Q$.

Proof: From the Lemma 66 we can say that in a fair execution of A, if P *ensures* Q *with probability one*, then the lemma holds. Otherwise, if (P *ensures* P_2 *with probability one*) $\wedge$ (P_2 *ensures* P_3 *with probability one*) $\wedge \cdots \wedge$ (P_{k-1} *ensures* Q *with probability one*), then by the same lemma we know that if $s_i \in P = P_1$ then with probability one there exists a $j_2 \geq i$ such that $s_{j_2} \in P_2 \Rightarrow$ with probability one $\exists j_3 \geq j_2$ such that $s_{j_3} \in P_3 \Rightarrow \ldots \Rightarrow$ with probability one $\exists j_k \geq j_{k-1}$ such that $s_{j_k} \in P_k = Q$, and the lemma holds. ■

Lemma 68 *Suppose A satisfies* P ensures Q, Q unless $(P \cup R)$, *and there exists a class* $C \in$ *part*(A) *such that for all states* $s \in Q$ *there is some step* $(s, \pi, s') \in$ *steps*(A) *such that* $s' \in R$, $\pi \in C$, *and* $p(s, \pi, s') \geq \epsilon > 0$. *Then for any fair execution* $\alpha = s_0 \pi_1 s_1 \pi_2 \ldots$ *of A, if* $s_i \in P$ *then with probability one there exists an* $m \geq i$, *such that* $s_m \in R$.

Proof: From lemma 4 we know that there must be some $k \geq i$ such that $s_k \in Q$. We also know that if α is a finite execution, by the fairness condition, no actions may be enabled in the final state, and since all the states in Q and P have enabled actions, then the end state must be in R. If α is infinite, let us suppose there is no such m. In that case, there are an infinite number of states in Q. We show this, by assuming there are only a finite number of states in Q. If this is so, then let s_l be the last such state. Since A satisfies Q *unless* $P \cup R$, then if s_{l+1}

is not in Q it must be in $P \cup R$, if s_{l+1} is in R, then there is a contradiction in our assumption, thus it must be in P, but since P *ensures* Q, then s_l could not have been the last state in Q and so again we find a contradiction, thus if there is no m such that $s_m \in R$ then there must be an infinite number of states in Q. For each state in Q there is a probability less than one of a transition to P. However, in order for there to be no state $s_m \in R$, this transition must be made an infinite number of times, (since there must be an infinite number of states in Q) thus the probability of such an execution is zero and thus with probability one there exists an $m \geq i$ such that $s_m \in R$. ■

We may use and alter some of the terms presented in section 2.10 to give us tools to reason about randomized I/O automata.

A *property chain with probability one*, $P_1, P_2, P_3, ..., P_n$, is a chain of distinct properties such that P_i *ensures* P_{i+1} *with probability one*, or in other words $P_i \overset{P_{i+1}P_{i+2}...P_{j-1}}{\longmapsto} P_j$ *with probability one*, for all $1 \leq i < j \leq n$.

Using this definition, we may make further observations about randomized automata.

Lemma 69 *Suppose A has a* property chain with probability one, $P_1, P_2, P_3, ..., P_n$, *A satisfies* P_n unless $P_1 \cup Q$ *and there exists a class,* $C \in part(A)$ *such that from every state in* P_n *there exists a step* $(s, \pi, s') \in steps(A)$ *such that* $s \in P_n$, $s' \in Q$, $\pi \in C$, *and* $p(s, \pi, s') > 0$. *Then in any execution* $\alpha = s_0\pi_1 s_1 \pi_2 ...$ *of A, if* $s_i \in P_1 \cup P_2 \cup ... \cup P_n$ *then there exists a* $k \geq i$ *such that* $s_k \in Q$ with probability one.

Actually, Lemma 68 is a special case of Lemma 69 and the proofs are similar.

5.3 A Simple Example : Coin Flip

A possible randomized I/O automaton that represents a coin being flipped follows:

- $vars(A)$ = {coin = {H,T}}
- $acts(A) = out(A)$ = {flip-heads, flip-tails}
- $start(A)$ = {H, T}
- $states(A)$ = {H, T}

- *steps*(A) = { ((H, flip-heads, H), .5), ((H, flip-tails, T), .5), ((T, flip-heads, H), .5), ((T, flip-tails, H), .5) }
- *part*(A) = {{flip-heads, flip-tails}}

It is easy to see here that A satisfies H *unless* T, but it does not satisfy H *ensures* T because not all actions π in the single class in *part*(A) satisfy $\{H \cap \sim T\}\pi\{T\}$ even though it is easy to see that with probability one if there is some state in a fair execution where H holds, there is some later state in that execution where T holds. However, A does satisfy H *ensures* T *with probability one*, which demonstrates a crucial proof step of progress properties of randomized algorithms.

5.4 Rabin-Lehmann's Randomized Dining Philosophers

Rabin and Lehmann have a randomized solution to the Dining Philosophers problem. It involves each processor deciding at random to decide which fork (left or right) it picks up first. This removes the symmetry of the problem that makes it unsolvable. The algorithm as presented in [LR] follows:

```
WHILE TRUE
  DO think;
    DO trying:=TRUE OR die OD;
    WHILE trying
      DO draw a random element s of {Right, Left};
           *** with equal probabilities ***
        WAIT until s chopstick is down
          AND THEN lift it;
        IF R(s) chopstick is down
          THEN lift it;
            trying:=false
          ELSE
            put down s chopstick
```

```
      FI
    OD;
  eat;
  put down both chopsticks
      *** one at a time, in an arbitrary order ***
OD.
```

where the function R is the reflection function on Right, Left.

We shall examine a slightly modified version where after eating, both chopsticks are put down simultaneously. The following is a randomized I/O automata representation of such a process p_i:

- $vars(A) = \{$
  - $state \in \{thinking, hungry, eating\}$,
  - f_i (left fork), f_{i+1} (right fork) $\in \{Left, Right, Down\}$,
  - $draw \in \{L, R\}$
  - $used - draw \in \{T, F\}\}$
- $states(A) = \prod_{v \in vars(A)} X_v$
- $start(A) = \{(thinking, Down, Down, L, T)\}$
- $in(A) = \{pickup - left_{i+1}, pickup - right_{i-1}, putdown - left_{i+1}, putdown - right_{i-1}, finish - eating_{i+}$
- $out(A) = \{pickup - left_i, putdown - left_i, pickup - right_i, putdown - right_i, finish - eating_i\}$
- $int(A) = \{draw\}$
- $part(A) = \{local(A)\}$
- $steps(A) = \{$

  * local action steps *

  ** steps of action *draw* from thinking **

  Effect: set draw to L or R each with probability 1/2,

reset used-draw to False

$\{((thinking, f_i \in \{L, D\}, f_{i+1} \in \{R, D\}, draw, used - draw), draw,$

$(hungry, f_i, f_{i+1}, R, F), .5)\}$

$\{((thinking, f_i \in \{L, D\}, f_{i+1} \in \{R, D\}, draw, used - draw), draw,$

$(hungry, f_i, f_{i+1}, L, F), .5)\}$

** steps of action *draw* from hungry **

Effect: draw = L or R each with probability 1/2

reset used-draw to False

$\bigcup\{((hungry, f_i \in \{L, D\}, f_{i+1} \in \{R, D\}, draw, T), draw, (hungry, f_i, f_{i+1}, L, F), .5)\}$

$\bigcup\{((hungry, f_i \in \{L, D\}, f_{i+1} \in \{R, D\}, draw, T), draw, (hungry, f_i, f_{i+1}, R, F), .5)\}$

** steps picking up first fork **

Enabled: if f_{draw} = Down (fork of draw is down)

Effect: sets $f_{draw} = opp(draw)$ (picks up fork of draw),

set used-draw to True

$\bigcup\{(hungry, D, f_{i+1} \in \{R, D\}, L, F), pickup - left_i, (hungry, R, f_{i+1}, L, T)\}$

$\bigcup\{(hungry, f_i \in \{L, D\}, D, R, F), pickup - right_i, (hungry, f_i, L, R, T)\}$

** steps picking up second fork **

Enabled: if $f_{opp(draw)}$ = Down (fork opposite of draw is down)

Effect: sets $f_{opp(draw)}$ = draw (picks up fork opposite draw)

$\bigcup\{(hungry, R, D, L, T), pickup - right_i, (eating, R, L, L, T)\}$

$\bigcup\{(hungry, D, L, R, T), pickup - left_i, (eating, R, L, R, T)\}$

** steps putting down first fork **

Enabled: $f_{opp(draw)} \neq$ Down (if second fork is not down)

Effects: $f_{draw} = D$ (puts down first fork)

$\bigcup\{(hungry, R, R, L, T), putdown - left_i, (hungry, D, R, L, T)\}$

$\bigcup\{(hungry, L, L, R, T), putdown - right_i, (hungry, L, D, R, T)\}$

** steps of action $finish - eating_i$ (puts down both forks) **

Enabled: when state = eating

Effects: $f_i, f_{i+1} = D$, state = thinking

$\bigcup\{(eating, f_i, f_{i+1}, draw, T), finish - eating_i, (thinking, D, D, draw, T)\}$

* input action steps *

** steps of action $pickup - left_{i+1}$ **

Effects: $f_{i+1} = R$ (right neighbor picks up fork)

$\bigcup\{(state \in \{thinking, hungry\}, f_i, f_{i+1}, draw, used - draw), pickup - left_{i+1},$

$(state, f_i, R, draw, used - draw)\}$

$\bigcup\{(eating, f_i, f_{i+1}, draw, used{-}draw), pickup - left_{i+1}, (thinking, D, R, draw, used{-}draw)\}$

(should never happen)

** steps of action $pickup - right_{i-1}$ **

Effects: $f_i = L$ (left neighbor picks up fork)

$\bigcup\{(state \in \{thinking, hungry\}, f_i, f_{i+1}, draw, used - draw), pickup - right_{i-1},$

$(state, L, f_{i+1}, draw, used - draw)\}$

$\bigcup\{(eating, f_i, f_{i+1}, draw, used{-}draw), pickup - right_{i-1}(thinking, L, D, draw, used{-}draw)\}$

(should never happen)

** steps of action $putdown - right_{i-1}$ **

Effects: $f_i = D$ (left neighbor puts down fork)

$\bigcup\{(state, f_i, f_{i+1}, draw, used{-}draw), putdown - right_{i-1}, (state, D, f_{i+1}, draw, used{-}draw)\}$

** steps of action $finish - eating_{i-1}$ **

Effects: $f_i = D$ (left neighbor puts down fork)

$\bigcup\{(state, f_i, f_{i+1}, draw, used{-}draw), finish - eating_{i-1}, (state, D, f_{i+1}, draw, used{-}draw)\}$

** steps of action $putdown - left_{i+1}$ **

Effects: $f_{i+1} = D$ (right neighbor puts down fork)

$\bigcup\{(state, f_i, f_{i+1}, draw, used{-}draw), putdown - left_{i+1}, (state, f_i, D, draw, used{-}draw)\}$

** steps of action $finish - eating_{i+1}$ **

Effects: $f_{i+1} = D$ (right neighbor puts down fork)

$\bigcup\{(state, f_i, f_{i+1}, draw, used{-}draw), finish - eating_{i+1}, (state, f_i, D, draw, used{-}draw)\}$

}

A fork f_i equals R if it is held by the philosopher on its right, p_i, it equals L if it is held by the philosopher on its left, p_{i-1}, and it equals D if it is down (held by neither). Fork $f_i = f_L$ for processor p_i and $f_{i+1} = f_R$ for processor p_i. The expression $opp(draw)$ represents the opposite side of the draw. State indicates what state the philosopher is in. Draw holds the value of the last draw. Used-draw is a boolean variable that is set to false when the draw is reset and set to true when the draw has been "used", in other words, since a new draw indicates what fork the philosopher will wait for (and thus eventually pick up) first, used-draw is set to true once this fork has been picked up and the draw used. Therefore used-draw could be considered as equal to the negation of the predicate "waiting to pick up the first fork".

We compose N such p_i using the SAJ-composition operator and call the composition A_c. We use the SAJ-composition operator rather than the composition operator defined in [LT] because we have represented the forks as shared variables and the states of the resulting automaton from an SAJ-composition are simpler in the case of shared variables. Note that the $i+1, i-1$ arithmetic is mod$N$ so as to form a circle of philosophers.

Let us call H the set of states where there exists a philosopher that is hungry and let us call E the set of states where some philosopher is eating. We would like to show that for any execution $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A_c $s_i \in H \Rightarrow$ there exists a $k \geq i$ such that $s_k \in E$ *with probability one.*

Let H_i be the set of states of A_c where $state(p_i) = hungry$. Let E_i be the set of states of A_c where $state(p_i) = eating$. Let D_i^R be the set of states of A_c where $draw(p_i) = R$. Let D_i^L be the set of states of A_c where $draw(p_i) = L$. Let $UD_i = used - draw(p_i)$.

First we shall present a proof with the same approach as that in [LR] but using the properties defined in this paper whenever possible. Let *dead* be the set of executions where there exists an $s_i \in H$ and for all $j \geq i$ $s_j \in H \cap \sim E$, in other words, the set of executions where someone is hungry at some state and at no later state does anyone eat.

Claim 70 *In any execution in dead, there are infinitely many fork pickups.*

Proof: We will show this claim to be true as a result of the following lemma.

The following lemma will show that a fork that is held is eventually put down.

Lemma 71 *In any execution of A_c, $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A_c, if $s_i \in (f_i \neq D)$, then there exists a $k > i$ such that $s_k \in (f_i = D)$*

Proof: There are two cases to consider: for the process p_i which has picked up the fork either it is the first fork (i.e. $f_i = f_{draw}$) or it is the second fork (i.e. $f_i = f_{opp(draw)}$).

Case 1: The only reachable states where f_{draw} (the first fork) is picked up are in $H_i \cap UD_i \cap (f_{draw} = opp(draw))$. Let us call this set of states P_i^1. By the definition of A_c, P_i^1 *ensures* $(f_{draw} = D) \cup E_i$. We also know that E_i *ensures* $(f_{draw} = D)$. Therefore by Lemma 4, this lemma holds.

Case 2: The only reachable states where $f_{opp(draw)}$ is picked up are in E_i. We know that E_i *ensures* $(f_{draw} = D)$. Therefore by Lemma 4, this lemma holds. ■

Suppose there are not an infinite number of fork pickups in an execution of *dead.* After the last fork pickup, eventually all the forks get put down by the previous lemma. Since by the definition of *dead* there are processes still in H. Therefore, they will still draw again and thus are guaranteed to pick up a fork by the definition of A_c. ■

The following lemma shows that if the draw of a hungry process has not yet been used, it will eventually be used (since the processor waits until the fork of the draw is down to pick it up) and that fork will be picked up.

Lemma 72 *In any execution of A_c, $\alpha = s_0\pi_1 s_1\pi_2 \ldots$ of A_c, if $s_i \in H_i \cap D_i^{draw} \cap \sim UD_i$, then there exists a $k > i$ such that $s_k \in (H_i \cap D_i^{draw} \cap UD_i \cap f_{draw} = opp(draw))$ and $s_{k-1} \notin (H_i \cap D_i^{draw} \cap UD_i \cap f_{draw} = opp(draw))$ $(draw \in \{L, R\})$. Furthermore, π_{k-1} = pickup-draw.*

Proof: By the definition of A_c and the previous lemma, we know that $H_i \cap D_i^{draw} \cap \sim UD_i \cap (f_{draw} \neq D)$ *ensures* $H_i \cap D_i^{draw} \cap \sim UD_i \cap (f_{draw} = D)$. By the definition of A_c and *ensures*, A_c satisfies $H_i \cap D_i^{draw} \cap \sim UD_i \cap (f_{draw} = D)$ *ensures* $(H_i \cap D_i^{draw} \cap UD_i \cap f_{draw} = opp(draw))$. We also know that the only action that results in this transition is *pickup-draw.* By Lemma ensures:l1 we know that there exists a $k \geq i$ such that $s_k \in (H_i \cap D_i^{draw} \cap UD_i \cap f_{draw} = opp(draw))$. Therefore by the definition of s_k (it is the first state where the fork is picked up), π_{k-1} must equal *pickup-draw.* We know that $k \neq i$ because the two properties are disjoint. ■

Claim 73 *If p_i draws infinitely often, then with probability one it will choose L infinitely often and R infinitely often.*

Proof: If p_i draws infinitely often, then p_i must be in H forever since *draw* is only enabled from H. By the definition of A_c, $H \cap D_i^{draw}$ *ensures* $(H \cap D_i^{opp(draw)}) \cup E_i$ *with probability one.* This means that if a philosopher is hungry with probability one he will eventually change the value of his last draw if he remains hungry. The claim is implied by the previous statements. ■

Lemma 74 *Let p and q be neighbors. If p picks up a fork infinitely often and q down not, then with probability one, p eats an infinite number of times.*

Proof: Let us consider the state s_i after which q does not pick up a fork. From Lemma 71 we know that any fork that q has at state s_i will be put down. Let us define s_j $(j \geq i)$ to be the state where q does not have the fork it shares with p and after which q does not pick up forks. Let us assume that q is the right neighbor of p. From the previous claim we know that with probability one p draws L infinitely often. When p draws L, since by the definition of A_c, H_i *ensures* $(H_i \cap D_i^L \cap \sim UD_i) \cup E_i$ *with probability one* and by Lemma 72, we are guaranteed of reaching some later state in $H_i \cap D_i^L \cap UD_i \cap f_L = R$. In other words, we are guaranteed that p will pick up its left fork at some later state. At this later state, since q does not have the right fork of p and $H_i \cap D_i^L \cap UD_i \cap (f_L = R) \cap (f_R = D)$ ensures E_i, p will eat. This will happen every time p draws L. ■

Lemma 75 *In any execution in dead, every process picks up a fork an infinite number of times with probability one.*

Proof: This follows from Lemma 74 and Claim **??**. ■

Let us define a *good configuration state* as one in $D_i^L \cap D_{i+1}^R$.

The following lemma states that from a state that is not a good configuration state, we are guaranteed to reach a good configuration state if all processes pick up forks infinitely often.

Lemma 76 *If every process picks up forks infinitely often, then with probability one, there are infinitely many occurances of good configuration states.*

Proof: It is easy to see that in a fair execution of A_c, if $s_i \in H_i \cap H_{i+1} \cap \sim(D_i^L \cap D_{i+1}^R)$ then with probability one there exists a $j > i$ such that $s_j \in (H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R) \cup E_i \cup E_{i+1}$ *with probability one.* This is a result of Lemma 68 where P is $H_i \cap H_{i+1} \cap \sim(D_i^L \cap D_{i+1}^R)$, Q is

$(D_i^R \cap D_{i+1}^L \cap H_i \cap H_{i+1}) \cup E_i \cup E_{i+1}$ and R is $(H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R) \cup E_i \cup E_{i+1}$. Therefore if p_i and p_{i+1} are not in a good configuration and remain hungry at some state in a fair execution s_i, then we are guaranteed with probability one that there exists some later state s_j $j > i$ such that $s_j \in H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R$ or $s_j \in E_i \cup E_{i+1}$ ■

Lemma 77 *For any fair execution, $\alpha = s_0\pi_1 s_1 \pi_2 \ldots$ of A_c, if $s_i \in H$ then with probability one there exists a $j \geq i$ such that $s_j \in E$.*

Proof: It is clear that $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1} \mapsto E_i \cup E_{i+1}$. In other words, if p_i and p_{i+1} are in a good configuration, are hungry, and have both not yet used their draw, one of them is guaranteed to eat. This is easily seen because $\sim UD$ *ensures* UD for all p_i (each unused draw will be used) so $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1} \mapsto H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap (UD_i \cap UD_{i+1}) \cap (f_i = R) \cap (f_{i+1} = L)$ meaning that a hungry and good configuration leads to the left philosopher holding the left fork and the right philosopher holding the right fork. Finally, $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap (UD_i \cap UD_{i+1})(f_i = R) \cap (f_{i+1} = L) \mapsto E_i \cup E_{i+1}$ meaning if the left philosopher is holding the left fork and the right philosopher is holding the right and they are in a good configuration and they are hungry, then one of them will eat which is obvious because there is no way both philosophers can be blocked to the middle fork.

From our definition of p_i, we know that $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R$ *ensures* $(H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1}) \cup \sim(D_i^L \cap D_{i+1}^R) \cup E_i \cup E_{i+1}$ which translates to a hungry and good configuration ensures either one philosopher eating or both philosophers still hungry and either a nongood configuration or a good configuration with both draws unused (thus guaranteeing someone eating from a previous argument). There exists a class such that for every state in $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R$ there is a positive probability step which results in a state in $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1} \cup E_i \cup E_{i+1}$. Therefore by Lemma 68 we know that if we have a state $s_l \in H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R$, then with probability one there exists an $m \geq l$ such that $s_m \in H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1} \cup E_i \cup E_{i+1}$.

From before we know that $H_i \cap H_{i+1} \cap D_i^L \cap D_{i+1}^R \cap \sim UD_i \cap \sim UD_{i+1} \cup E_i \cup E_{i+1}$ *ensures* $E_i \cup E_{i+1}$. Therefore the lemma holds. ■

Chapter 6

Conclusion

This paper has sought to bring together the concepts of Chandy and Misra's UNITY proof system and Lynch and Tuttle's I/O automaton model, a more general computational model than that of UNITY due to its classification of actions and local action partition feature. A mapping from UNITY programs to UNITY automata, a subset of I/O automata, is defined. The UNITY proof concepts and composition operators are generalized and adapted to I/O automata. Furthermore, examples have been presented illustrate this approach to reasoning about I/O automata.

UNITY proof concepts are useful to reason about I/O automata (as well as UNITY programs) because they represent safety and progress properties in a straightforward way. Specifically, ensures, leads to, and until are progress properties that can guarantee the existence of a future state in a certain set given that the current state is in another particular set of states. Such a property of an I/O automaton is the essence of progress. It is also easy to see if any of these UNITY properties is satisfied by an I/O automaton. Similarly, safety properties, such as Fixed Point, invariant, stability, and even unless in certain circumstances are easy to check for satisfaction and using some of the lemmas can insure certain characteristics in all fair executions of an I/O automaton. Of course, the UNITY properties are not by any means a complete set of the interesting properties of programs or automata. Further work in defining other such properties to facilitate progress or safety proofs about I/O automata I feel would be rewarding.

A set of composition operators were defined here and proofs presented regarding the properties of the composed automata based on the properties of its components. There is a wide

spectrum of composition operators that can express useful combining of I/O automata, or even decomposition of I/O automata. In [CM], often properties of a program were proven using the fact that it was a composition of some other programs. Perhaps, some kind of [de]composition operator could help prove interesting properties about I/O automata or its components.

Finally, an augmentation of the I/O automaton model and the UNITY proof system has been presented to aid in reasoning about randomized algorithms. I feel even more could be added to this model by applying the analysis of discrete-state discrete-transition Markov Processes (and continuous transition for the continuous randomized case) to analyze the probabilities between sets of states in a randomized I/O automaton. This method could possibly be generalized and put in terms of properties similar to ensures and leads to. The biggest problem with the properties defined by [CM] is that they are very dependent on the definition of the state transitions (or actually, more accurately the assignments). Using limiting state probability theory would be useful in avoiding problems due to this and allow reasoning about randomized I/O automata at a higher level of abstraction.

An augmentation to the I/O automaton analogous to that of the randomized I/O automata in that it adds one component to "keep track of" the property of interest, in this case time, has resulted in the timed I/O automaton model [MMT]. I feel that UNITY proof concepts expanded in a way similar to the expansion for the randomized I/O automata can be used to aid in some time bounds analyses and proofs about timed I/O automata. I had started on such work but had not made sufficient progress to include a chapter in this thesis. The following could indicate a starting point for such an application of UNITY proof concepts:

Lemma 78 *If A satisfies* P unless Q *then in any timed execution* $\alpha = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\ldots$ *of A, if* $s_i \in P \cap \sim Q$ *and either* $i = 0$ *or* $s_{i-1} \notin P$, *then there does not exist a* $j > i$ *with* $t_j < t_i + \ell$ *and* $\pi_j \in Q$, *where* $\ell = \min(C_l : \exists(s, \pi, s') \in steps(A)$ *such that* $\pi \in C$, $s \in P$, *and* $s' \in Q)$.

Proof: From definition of Timed I/O automata and the definition of unless. ■

Lemma 79 *If A satisfies* P ensures Q *then in any timed execution* $\alpha = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\ldots$ *of A, if* $s_i \in P \cap \sim Q$, *then there exists a* $j > i$ *with* $t_j \leq t_i + C_u$ *such that* $s_j \in Q$.

Proof: From the definition of Timed I/O automata and the definition of ensures. ■

Lemma 80 *If A satisfies* P $\mapsto$ Q *then in any timed execution* $\alpha = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\ldots$ *of A, if* $s_i \in P \cap \sim Q$, *then there exists a* $j > i$ *with* $t_j \leq t_i + C_U$ *such that* $s_j \in Q$, *where* $C_U = \sum_{i=1}^{k-1} C_u(P_i)$ *where* P_i *denotes the set of states from the definition of leads to* ($\mapsto$): *there exists a sequence* $P_1, \ldots P_k$, $(k \geq 2)$ *of sets where* $P_1 = P$, $P_k = Q$, *and A satisfies* P_i ensures P_{i+1} *for all* $1 \leq i \leq k-1$.

Proof: From the definition of Timed I/O automata and the definition of leads to. ■

Furthermore some of the state definitions (transient, etc) presented in the chapter about randomized automata may be useful in determining some execution time bounds.

6.1 Errors Found In [CM]

There were two errors in [CM] that I noted during the research phase of this paper:

1. the error in [CM] in the example of superposition on page 166. They set out to show that by applying superposition, a superposed program can be found such that *p detects q* where q is a predicate on the variables of the lower level program. They define a superposition. A property W is defined as the number of statement executions in the underlying program. This is the q to be detected. However, q is not a predicate on the variables of the lower level program.

2. the error in CM page 257 in their definition of *partial.* I believe the cases are switched for *x.done* holding and not holding. Their explanation and example is consistent with the switch.

Bibliography

[CL] Chandy, K.M., Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM TOCS,* 3:1, February 1985, pp 63-75.

[CM] Chandy, K.M., and Misra, J. *A Foundation of Parallel Program Design.* Addison-Wesley, 1988.

[LR] Lehmann, D. Rabin, "On The Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem," *ACM* 0-89791-029-X/81/0100-0133, 1981.

[LT] Lynch, N.A., and Tuttle, M.R. "Heirarchical Correctness Proofs for Distributed Algorithms," Master's Thesis, Massachusetts Institute of Technology, April, 1987. MIT/LCS/TR-387, April 1987.

[MMT] Merritt, M., Modugno, F., and Tuttle, M. "Time Constrained Automata." Manuscript.