Dynamic Process Creation in a Static Model

by

John Leo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1990

© John Leo, 1990

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 1990

Certified by . . . .
Nancy Lynch
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 10 1990
LIBRARIES

Dynamic Process Creation in a Static Model

by

John Leo

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1990, in partial fulfillment of the
requirements for the degrees of
Master of Science
and
Bachelor of Science

Abstract

Distributed algorithms that involve dynamic process creation and changing topologies are modeled using I/O automata, a static model. Two examples of such algorithms are proven correct. In addition, the Actor model, for which process creation is a major component, is given a rigorous basis using I/O automata.

Proof techniques are developed and demonstrated, including a simple variant function technique for liveness proofs. Rely/guarantee functions are developed for I/O automata, and it is shown how modular proofs of correctness can be achieved by using specifications based upon them.

Thesis Supervisor: Nancy Lynch
Title: Professor of Computer Science and Engineering

Acknowledgements

The idea for this thesis is due to Dennis de Champeaux at HP Labs, and the examples of Set Partition and Olympic Torch were suggested by him. Our paths have diverged, and his work can be found in [dC89]. Reed Letsinger, Mike Lemon, Ivan Tou, Wendy Fong, Michelle Lee, Bill Stanton, Pierre Huyn, Warren Harris, Allan Shepherd, Allan Kuchinsky and many other people made HP Labs a fun place to work.

At MIT, Mark Tuttle, Hagit Attiya, Ken Goldman and Steve Ponzio offered helpful comments and discussions. Mark in particular taught me everything about I/O automata I know. Carl Hewitt helped me understand actors. Particularly I want to thank my thesis supervisor Nancy Lynch, who carefully read the thesis, offered many good comments, and finally allowed me to graduate.

Nintendo, Dan Nickolich, David Watson, Tomoko Graham, Akiko Yano, Kinyobi ni wa hana o katte, Alain Robbe-Grillet, etc., made the years spent working on this thesis worthwhile. But most of all I want to thank Kyoko Watanabe, for all those things which cannot be put into words.

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and not necessarily the views of the National Science Foundation.

Contents

1 Introduction **6**

2 Preliminaries **9**

2.1 Some Definitions and Notation 9

2.2 I/O Automata 10

2.2.1 Basic Definitions 10

2.2.2 Composition 11

2.2.3 Action Hiding and Renaming 13

2.2.4 Fairness 14

2.2.5 Possibilities Mappings 16

2.3 Variant Functions 17

2.4 Rely/Guarantee Functions 18

2.4.1 Proof Techniques 20

3 Process Creation **25**

3.1 Definitions 25

3.2 Abstraction and Rely/Guarantee Functions 26

3.3 Comments 27

4 Set Partition **28**

4.1 Specification 28

4.2 Deterministic Sequential Implementation 30

4.3 Distributed Solution with Instantaneous Message Passing 32

4.4 Comments and Comparisons 40

5 Olympic Torch 42

5.1 Description of Olympic Torch 42

5.2 Parallel Olympic Torch 43

5.3 Correctness 47

5.4 Comments 51

6 Actors 53

6.1 Example: Recursive Factorial 54

6.2 Agha's Operational Semantics for Actors 56

6.2.1 Basic Definitions 56

6.2.2 Actor Behaviors 58

6.2.3 Transitions 59

6.3 I/O Automata Actors 61

6.3.1 I/O Automata Actors with Separated Mail System 61

6.3.2 Simplified I/O Automata Actors 64

6.3.3 Correspondence between I/O Automata Actors and Agha's Model 66

6.4 Correctness Proofs 67

6.4.1 Recursive Factorial Revisited 68

6.4.2 Comments 69

7 Conclusions 72

Chapter 1

Introduction

Proofs of correctness for distributed systems have generally concentrated on static systems, in which there is a predetermined set of processes and a fixed topology. Numerous proof systems have been developed to prove correctness of algorithms in such a setting; see the introduction of [LT87] for a survey. However none of these models explicitly addresses the issue of process creation. How can one prove correctness of a distributed system in which the set of processes, as well as the communication topology, may change over time?

One solution is to create a new model which supports dynamic process creation, and attempts at this were made by the author and Dennis de Champeaux, but none were satisfactory. This work evolved into [dC89]; problems with this approach will be discussed later. Another approach is to use an existing system and add tools for process creation to it. This is what is done in this thesis. The proof system chosen was Lynch and Tuttle's I/O automaton model ([LT87, LT88]), which although young has already been used to prove correctness for a variety of algorithms (see the latter paper for a partial list). The I/O automaton model is described in Chapter 2. Along with being a very general model it has a number of nice features for correctness proofs. One can specify modules at various levels of abstraction, and also do hierarchical proofs in which the entire algorithm is modeled at successively more detailed levels of abstraction. One can not only write specifications as I/O automata; one can also write the algorithms themselves as automata. In fact a programming system called Spectrum is being developed based upon I/O automata by Goldman ([Gol90]). Versions of the examples in this thesis were programmed and simulated using Spectrum.

Adding process to creation to I/O automata is rather simple. The model allows there to be an infinite number of automata; one simply defines a predicate for each automaton which is true if and only if the automaton is "alive." Furthermore such an automaton must satisfy certain properties: in particular if it is not alive it cannot send any messages or change state. A similar method could be used for any model in which an infinite number of processes is allowed. Although this method is simple, one might still worry how difficult it is to do proofs of algorithms involving process creation in such a model. Two examples are presented in this thesis: Set Partition and Olympic Torch. In Set Partition only two processes are created so this example isn't very different from a static algorithm. Olympic Torch, on the other hand, is almost purely concerned with process creation and a changing topology. As will be seen, the proofs of these two algorithms are not overly complicated. And what might be surprising is that the proof of Set Partition was more difficult than that for Olympic Torch. This suggests that process creation is not the major factor of difficulty; more important is the structure of the algorithm itself.

There does not appear to be one best technique to prove correctness for algorithms involving dynamic process creation; rather just as for static algorithms the proof should depend upon what structure of the algorithm can be exploited. However a proof technique that seems quite useful is that of *rely/guarantee conditions.* There are numerous versions of this technique in the literature ([MC81, Jon83, Sta84, AL90]) but the basic form is the same. One specifies a process by what it assumes about the environment and the corresponding guarantees it makes about its outputs. One can then combine the rely/guarantee conditions for a set S of processes by showing for each process $P \in S$ that the guarantees of the environment of S along with the guarantees of each process in $S - \{P\}$ satisfy the assumptions of P. This technique has some intuitive appeal for process creation since when a process creates children it can be reasoned that the children assume certain conditions about the parent and in return provide certain guarantees to the parent. Rely/guarantee conditions are defined in terms of functions in Section 2.4 and are used to prove correctness for both Set Partition and Olympic Torch.

Dennis de Champeaux's work ([dC89]) is the closest to this one. He was also primarily concerned with process creation and uses the same two examples. However there are numerous problems with his approach. He never defines a rigorous underlying model, so it is never clear

what it means for an algorithm to be "correct," and it is unclear what the proofs actually show. Most surprising, despite the apparent emphasis on process creation, process creation itself is never handled! That is, no distinction is made between a system in which processes are created and die dynamically and a system in which all processes are alive all the time. Specific problems with his proofs of Set Partition and Olympic Torch will be discussed in their respective chapters. This work still seems very preliminary.

Having done two examples, we next look at a programming methodology which makes extensive use of dynamic process creation and changing topologies, namely Carl Hewitt's Actors ([Agh86, Cli81]). This dynamic nature of Actors led Agha to claim that it is a more powerful model than static models; however we will show that this is not true. Agha gives a model for actors in his book, and we will show that a model for actors defined in I/O automata is equivalent to it. In addition by doing this we were able to fix several poorly defined or vague notions in Agha's model. Correctness proofs have apparently not been done in an actor model since Yonezawa's work ([Yon77]). Having given Agha's model a rigorous basis, we give a proof for a simple example. It seems proofs for more complicated examples will be quite difficult to do due to some peculiarities of the model, however; this is discussed at the end of the chapter along with some possible solutions.

The organization of the thesis is as follows. In the Preliminaries, definitions and theorems for I/O automata are presented. Also a simple variant function for I/O automata is presented, as are rely/guarantee functions. Chapter 3 describes how process creation can be modeled in I/O automata. Then in the chapters for Set Partition and Olympic Torch the ideas in the previous chapters are used to prove correctness for these examples. Finally, in Chapter 6, actors are defined in terms of I/O automata, and a simple proof of correctness is presented.

Chapter 2

Preliminaries

The first section of this chapter contains notation that will be used throughout. The second section contains definitions and theorems concerning I/O automata. These two sections may be skipped now and returned to when particular information is needed.

Section 2.3 gives the definition of a simple variant function that can be used to prove liveness properties within the I/O automata model. This variant function is used to prove correctness for a sequential implementation of Set Partition in Chapter 4. Section 2.4 presents a method by which an I/O automaton can be specified using assumptions about the environment and the corresponding guarantees the automaton will make based upon those assumptions. It also includes two theorems about action hiding that are useful when using rely/guarantee functions.

2.1 Some Definitions and Notation

If S is a set and a is some element, then $S + a = S \cup \{a\}$ and $S - a = S - \{a\}$.

A *sequence* of elements from set S is an ordered list of those elements. The empty sequence is denoted ϵ. Given a non-empty sequence $s = s_1 s_2 s_3 \ldots$, head$(s) = s_1$ and tail$(s) = s_2 s_3 \ldots$. If s is finite then $|s|$ denotes the length of s and last$(s) = s_{|s|}$. If $t = t_1 \ldots t_m$ is a sequence of elements from S and $x \in S$ then $t \cdot s = t_1 \ldots t_m s_1 \ldots$ and $t \cdot x = t_1 \ldots t_m x$. For s finite, $s \sqsubseteq t$ means s is a prefix of t (i.e., there exists some sequence u such that $t = s \cdot u$). For a sequence s, $\{s\}$ indicates the multiset having the same elements as s. Occasionally we will write s_i as $s(i)$, or $s_1 s_2 s_3 \ldots$ as $[s_1, s_2, s_3, \ldots]$ for clarity.

The notation (a, b) is used for a pair of objects a and b. Then $(a, b) = (c, d)$ means $a = c$ and $b = d$. Similarly $(a, b) \neq (c, d)$ means that $a \neq c$ or $b \neq d$.

2.2 I/O Automata

2.2.1 Basic Definitions

We briefly review the fundamentals of I/O Automata. For the details and proofs, see [LT88, LT87]. An *I/O Automaton A* is a tuple consisting of:

- an action signature sig(A), a partition of the actions acts(A) into three classes: in(A), out(A) and int(A); the input, output, and internal (respectively) actions of A,
- *states*(A), the states of the automaton,
- a nonempty set of *start states* of A, start(A) $\subseteq$ states(A),
- a transition relation steps(A) $\subseteq$ states(A) $\times$ acts(A) $\times$ states(A),
- an equivalence relation part(A) partitioning the set local(A) = out(A)$\cup$int(A) (the *locally-controlled actions*) into at most a countable number of equivalence classes.

An action π is *enabled* from state a if there exists some state a' such that $(a, \pi, a') \in$ steps(A). Every I/O Automaton A must satisfy the requirement that all input actions are enabled from every state of A.

The *external actions* of automaton A are ext(A) = in(A) $\cup$ out(A). An *external action signature* is an action signature having only external actions.

An *execution fragment* of an automaton A is a finite sequence $a_0\pi_1a_1\pi_2\ldots\pi_na_n$ or an infinite sequence $a_0\pi_1a_1\pi_2\ldots$ of alternating states and actions of A such that $(a_i, \pi_{i+1}, a_{i+1}) \in$ steps(A) for all $i \geq 0$. An *execution* of A is an execution fragment such that $a_0 \in$ start(A). The set of executions of A is denoted by execs(A). Say that a state $a \in$ states(A) is *reachable* if a is in x for some $x \in$ execs(A). The *schedule* of an execution fragment x is the subsequence sched(x) = $\pi_1\pi_2\ldots$ of actions appearing in x. The set of schedules of A is the set of schedules of each $x \in$ *execs*(A) and is denoted scheds(A). If z is a schedule and Π is a set of actions

then $z|\Pi$ is the subsequence of z consisting only of actions in Π. If x is an execution, then $x|\Pi = \text{sched}(x)|\Pi$. If X is a set of schedules or executions then $X|\Pi = \bigcup_{x \in X} \{x|\Pi\}$.

We sometimes use an *execution module* to represent executions satisfying certain liveness conditions. An execution module E consists of a set states(E) of states, and action signature sig(E), and a set execs(E) of executions. An execution module E is said to be an execution module of an automaton A if states(E) = states(A), sig(E) = sig(A), and execs(E) $\subseteq$ execs(A). The execution module of A whose executions are execs(A) is denoted Execs(A).

An *schedule module* is sometimes used to represent correctness conditions. A schedule module S consists of an action signature sig(S) and a set scheds(S) of schedules. An *external schedule module* is a schedule module with an external action signature. For an execution module E, the schedule module with the same action signature and schedules as E is denoted Scheds(E), and Scheds(A) is defined to be Scheds(Execs(A)). An *object* is either an automaton, an execution module, or a schedule module. The external schedule module of an object O, denoted External(O), is defined to be the external schedule module with the external action signature of O and the schedules $\{z|\text{ext}(O) : z \in \text{scheds}(O)\}$. For a schedule z and object O, we often write $z|O$ for $z|\text{acts}(O)$.

Typically states of automata are defined as the cross product of several components. If $a = \Pi c_i$, then we denote component c_i of state a as $a.c_i$.

2.2.2 Composition

We only compose compatible automata. A set of action signatures $\{S_i : i \in I\}$ are *compatible* if for all $i, j \in I, i \neq j$, both $\text{out}(S_i) \cap \text{out}(S_j)$ and $\text{int}(S_i) \cap \text{acts}(S_j)$ are empty. This is because communication is modeled by shared actions, and we want only one process to control an action. This definition of compatibility is not strong enough for certain theorems, so we define a set a set of action signatures $\{S_i : i \in I\}$ to be *strongly compatible* if they are compatible and the following condition is also satisfied. Let $J \subseteq I$ be the set of indices j such that S_j contains an action that is contained in S_i for infinitely many $i \in I$. Let $K \subseteq J$ be the set of indices k such that $\text{int}(S_k) \neq \emptyset$. Then the additional condition for strong compatibility is that $|K|$ is finite. This is slightly weaker than the definition of strong compatibility in [LT88]. The condition there was that $J = \emptyset$; i.e., that no action is contained in infinitely many signatures.

Our condition is that for any action contained in infinitely many signatures only finitely many of these signatures may contain internal actions. Though not the weakest condition possible it is satisfactory and all theorems requiring the old definition of strong compatibility also hold under the new definition.

A set of automata $\{A_i : i \in I\}$ are *(strongly) compatible* iff their action signatures are (strongly) compatible. The composition $S = \Pi_{i \in I} S_i$ of compatible action signatures $\{S_i : i \in I\}$ is defined to be the action signature with:

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

To compose states, we use a Cartesian product. We represent the composite state as a where $a = \Pi_{i \in I} a_i$ and thus the ith component of a is a_i, sometimes written as $a|A_i$ ("a restricted to A_i"). The composition $A = \Pi_{i \in I} A_i$ of compatible automata $\{A_i : i \in I\}$ is defined to be the automaton with

- $\text{sig}(A) = \Pi_{i \in I} \text{sig}(A_i)$,
- $\text{states}(A) = \Pi_{i \in I} \text{states}(A_i)$,
- $\text{start}(A) = \Pi_{i \in I} \text{start}(A_i)$,
- $\text{part}(A) = \bigcup_{i \in I} \text{part}(A_i)$, and
- $\text{steps}(A)$ is the set of triples (a, π, a') where $a, a' \in \text{states}(A)$, $\pi \in \text{acts}(A)$, and for all $i \in I$, if $\pi \in \text{acts}(A_i)$ then $(a_i, \pi, a'_i) \in \text{steps}(A_i)$, otherwise if $\pi \notin \text{acts}(A_i)$ then $a_i = a'_i$.

Note that composition is both commutative and associative.

If $A = \Pi_{i \in I} A_i$, and $x \in \text{execs}(A)$ where $x = a_0 \pi_1 a_i \ldots$, then $x|A_i$ is the execution formed by deleting $\pi_j a_j$ whenever $\pi_j \notin \text{acts}(A_i)$ and replacing each of the remaining a_j with $a_j|A_i$. Note that it follows $x|A_i \in \text{execs}(A_i)$, and is essentially x projected onto A_i (this is Lemma 1 of [LT87]).

Execution and schedule modules are (strongly) compatible iff their actions signatures are (strongly) compatible. The composition $E = \Pi_{i \in I} E_i$ of compatible execution modules is is

defined so that states$(E) = \Pi_i$states(E_i) and sig$(E) = \Pi_i$sig(E_i). Given an alternating sequence $x = a_0\pi_1 a_1 \ldots$ of states and actions of E, define $x|E_i$ to be the sequence obtained by removing $\pi_j a_j$ if π_j is not an action of E_i and $a_j|E_i = a_{j-1}|E_i$, and replacing the remaining a_j by $a_j|E_i$.[1] Then execs$(E) = \{x : x|E_i \in$ execs(E_i) for all $i \in I\}$. The composition $S = \Pi_{i \in I} S_i$ of compatible schedule modules is similarly defined so that sig$(S) = \Pi_i$sig(S_i), and scheds$(S) = \{z : z|S_i \in$ scheds(S_i) for all $i \in I\}$.

2.2.3 Action Hiding and Renaming

Communication between I/O automata is modeled by shared actions. When we compose two automata we may want the communication between them to not be visible to the environment. Since composition does not hide communication we must do it explicitly. For an automaton A and a set of actions Σ such that $\Sigma \cap \text{in}(A) = \emptyset$, define the automaton Hide$_\Sigma(A)$ to be identical to A except that the action signature is now:

1. in(Hide$_\Sigma(A)$) = in(A)
2. out(Hide$_\Sigma(A)$) = out$(A) - \Sigma$, and
3. int(Hide$_\Sigma(A)$) = int$(A) \cup (\Sigma \cap \text{out}(A))$.

This is slightly different than the definition in [LT87] in that we only allow actions of local(A) to be hidden, because we want part(Hide$_\Sigma(A)$) = part(A). Using the previous definition part(Hide$_\Sigma(A)$) is not well defined if input actions are hidden. Since typically we want to hide internal communication (which due to composition will be initially an output action) this is not a serious restriction. There doesn't seem to be any use for hiding input actions.

The following is Lemma 12 of [LT87].

Theorem 2.1 *For all automata A, execution modules E, schedule modules S, and set of actions Σ,*

1. Execs($Hide_\Sigma(A)$) = Hide$_\Sigma$(Execs(A))

2. Scheds($Hide_\Sigma(E)$) = Hide$_\Sigma$(Scheds(E))

[1] This definition is slightly different from that of [LT87], in that one can only remove an action and the resulting state if this action does not effect the state with respect to E_i.

3. $\text{External}(Hide_\Sigma(S)) = \text{External}(\text{Hide}_\Sigma(\text{External}(S)))$

The following is Lemma 14 of [LT87].

Theorem 2.2 *Let* $\{O_i : i \in I\}$ *be a collection of compatible objects, and let* $\{\Sigma_i : i \in I\}$ *be a collection of sets of actions. If* $\text{acts}(O_i)$ *and* Σ_j *are disjoint for all* $i \neq j$*, then* $\text{Hide}_{\cup_i \Sigma_i}(\Pi_{i\in I} O_i) = \Pi_{i\in I}\text{Hide}_{\Sigma_i}(O_i)$.

In order to make two automata compatible we may need to rename some of the actions of one or both of the automata. This is done using an *action mapping* f, an injective mapping between sets of actions. A mapping f is *applicable* to an automaton A if $\text{acts}(A)$ is contained in the domain of f. The resulting automata $f(A)$ is the same as A except that all actions are appropriately renamed, and the transition relation and partition are modified accordingly.

The operations of hiding and renaming are commutative; that is for any automaton A and applicable action mapping f, $\text{Hide}_{f(\Sigma)}(f(A)) = f(\text{Hide}_\Sigma(A))$ ([LT87], Lemma 16).

2.2.4 Fairness

We want to guarantee that every process (where a process can be thought of as corresponding to the actions in one partition of an I/O Automaton) has a chance to take a step if one of its actions is enabled. The definition of *fairness* we use is sometimes called *weak fairness* and has the property that an implementation in which each process is on a separate processer is guaranteed to be fair as long as each processor has a nonzero finite running speed.

A *fair execution* of an automaton A is an execution x of A such that the following conditions hold for every $C \in \text{part}(A)$:

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

We say that action α is *continuously enabled* from state a if for all execution fragments $y = a\pi\ldots$ beginning with a which do not contain α, α is enabled from every state of y. A special case of the second condition is that if some action $\alpha \in C$ is continuously enabled from a and no other action of C is enabled more than finitely often in any y, then α must occur at some point after

a in any fair execution that contains a. Note however than an input action to an automaton, despite being continuously enabled, need not occur since it is not a local action.

Define fair(A) to be the set of fair executions of automaton A, and define Fair(A) to be the execution module of A having fair(A) as its set of executions. The *fair behavior* of A, denoted Fbeh(A), is defined to be the schedule module External(Fair(A)). For other objects O (execution modules and schedule modules), Fbeh(O) = External(O). For any object O the set of schedules of Fbeh(O) is denoted fbeh(O).

The following are Lemma 19 and Lemma 20 of [LT87]. Notice that they require strong compatibility.

Theorem 2.3 $\mathrm{Fair}(\Pi_{i\in I}A_i) = \Pi_{i\in I}\mathrm{Fair}(A_i)$ *for strongly compatible automata* $\{A_i : i \in I\}$.

Theorem 2.4 $\mathrm{Fbeh}(\Pi_{i\in I}O_i) = \Pi_{i\in I}\mathrm{Fbeh}(O_i)$ *for strongly compatible objects* $\{O_i : i \in I\}$.

Objects O and O' are called *fairly equivalent* if Fbeh(O) = Fbeh(O'). Object O is said to *solve* object O' if fbeh(O) $\subseteq$ fbeh(O') and O and O' have the same external action signature. Note that O cannot be a trivial solution since input actions are always enabled. Also note that "solves" is reflexive and transitive.

Theorem 2.5 *Let* $\{O_i : i \in I\}$ *and* $\{P_i : i \in I\}$ *each be sets of compatible objects, and let* $O = \Pi_{i\in I}O_i$ *and* $P = \Pi_{i\in I}P_i$. *If for all* $i \in I$, O_i *solves* P_i, *then* O *solves* P.

Proof: The external action signatures of O and P must be the same from the definition of composition. We must prove fbeh(O) $\subseteq$ fbeh(P). Let $z \in$ fbeh(O). Then for all $i \in I$, $z|O_i \in$ fbeh(O_i) by Theorem 2.4. Since the external action signatures of O_i and P_i are the same, $z|O_i = z|P_i$. Thus $z|P_i \in$ fbeh(P_i) for all $i \in I$ by hypothesis. Using Theorem 2.4 again it follows $z \in$ fbeh(P). □

Theorem 2.6 *Let* A *and* B *be I/O Automata. Let* $\Pi \subseteq \mathrm{out}(A)$ *be a set of actions. If* A *solves* B *then* $\mathrm{Hide}_\Pi(A)$ *solves* $\mathrm{Hide}_\Pi(B)$.

Proof: A and B must have the same external action signature and partition both before and after hiding. Let $A' = \mathrm{Hide}_\Pi(A)$ and $B' = \mathrm{Hide}_\Pi(B)$. We need to show fbeh($A'$) $\subseteq$ fbeh(B'). If $z \in$ fbeh(A'), then $z \in$ fbeh(A) since part(A') = part(A) and ext(A') $\subseteq$ ext(A). Therefore

$z \in \text{fbeh}(B)$ since A solves B, and $z \in \text{fbeh}(B')$ since $\text{part}(B) = \text{part}(B')$ and the $\text{ext}(A') = \text{ext}(B')$. □

2.2.5 Possibilities Mappings

An *extended step* of an automaton A is defined to be a triple of the form (a, β, a') where a and a' are states of A, and β is a finite sequence of actions of A such that $\beta = \text{sched}(x)$ for some execution fragment $x = a\pi_1 \ldots \pi_n a'$ of A. It's possible that $x = a$ in which case $a' = a$ and $\beta = \epsilon$. Let A and B be two automata with the same external action signature. Let h be a mapping from states(A) to $2^{states(B)}$ (the power set of states of B). The mapping h is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state a_0 of A, there is a start state b_0 of B such that $b_0 \in f(a_0)$.
2. If a is a reachable state of A, $b \in f(a)$ is a reachable state of B, and (a, π, a') is a step of A, then there is an extended step (b, γ, b') of B such that
   (a) $\gamma|\text{ext}(B) = \pi|\text{ext}(A)$, and
   (b) $b' \in f(a')$.

Possibilities mappings are used in hierarchical correctness proofs; once one has proven there is a possibilities mapping from A to B much of the work has been done showing that A solves B. The following are several useful theorems about possibilities mappings.

Let $x \in \text{execs}(A)$ and $y \in \text{execs}(B)$. Given a possibilities mapping h from A to B, we say that y *finitely corresponds* to x under h if x and y are both finite, $\text{sched}(y)|\textit{extern}(B) = \text{sched}(x)|\text{ext}(A)$, and $b_j \in h(a_i)$ where a_i and b_j are the final states of x and y respectively. We say that y *corresponds* to x under h if for every finite prefix $x_i = a_0\pi_1 \ldots \pi_i i$ of x there is a finite prefix y_j of y finitely corresponding to x_i under h such that y is the limit of the y_j.

The following is Lemma 28 from [LT87].

Theorem 2.7 *Let h be a possibilities mapping from A to B. If x is an execution of A, then there is an execution y of B corresponding to x under h.*

Theorem 2.8 *Let A, B and C be I/O Automata. If g is a possibilities mapping from A to B and h is a possibilities mapping from B to C then $h \circ g$ is a possibilities mapping from A to C.*

Proof: All the automata must have the same external action signature. For any start state s of A there is a start state t of B such that $t \in g(s)$. For t there is a start state u of C such that $u \in h(t)$. Thus $u \in h(g(s))$.

If a is a reachable state of A, $b \in g(a)$ is a reachable state of B, and (a, π, a') is a step of A, then there is an extended step (b, γ, b') of B such that $\gamma|\text{ext}(B) = \pi|\text{ext}(A)$ and $b' \in g(a')$ by definition. Let $x = b_0\gamma_1 b_1\gamma_2 \ldots b_{n-1}\gamma_n b_n$ the the execution fragment of B corresponding to the extended step (b, γ, b') where $b_0 = b$, $b_n = b'$, and each γ_i is either an internal action of B or π. If π is an external action of A (and thus also B) then π will appear exactly once in γ; otherwise π will not appear. For each step $(b_i, \gamma_{i+1}, b_{i+1})$ there is an extended step $(c_i, \delta_{i+1}, c_{i+1})$ such that $c_i \in h(b_i)$, $c_{i+i} \in h(b_{i+i})$, and $\delta_{i+1}|\text{ext}(C) = \gamma_{i+1}|\text{ext}(B)$. Let $\delta = \delta_0 \cdot \delta_1 \cdots \delta_n$. Then $\delta|\text{ext}(C) = \gamma|\text{ext}(B) = \pi|\text{ext}(A)$. Also $c_0 \in h(b)$ and thus $c_0 \in h(g(a))$. Similarly $c_n \in h(b')$ and therefore $c_n \in h(g(a'))$. It follows that $h \circ g$ is a possibilities mapping from A to C. □

Theorem 2.9 *Let A and B be I/O Automata. Let Π be a set of actions. If there is a possibilities mapping h from A to B then h is a possibilities mapping from* $\text{Hide}_\Pi(A)$ *to* $\text{Hide}_\Pi(B)$.

Proof: A and B must have the same action signature, so the theorem is immediate. Each action $\pi \in \Pi$ merely becomes an internal action if it wasn't already. □

The following is Lemma 31 of [LT87]. It says that if there are possibilities mappings between the automata composing A and B, then there is also a possibilities mapping between A and B.

Theorem 2.10 *Suppose for all $i \in I$ that h_i is a possibilities mapping from A_i to B_i, and that* $\text{acts}(B_i) \subseteq \text{acts}(A_i)$. *let $A = \Pi_i A_i$ and $B = \Pi_i B_i$. If h is the mapping from* $\text{states}(A)$ *to the power set of* $\text{states}(B)$ *defined by $h(a) = \{b : b|B_i \in h_i(a|A_i)\}$, then h is a possibilities mapping from A to B.*

2.3 Variant Functions

Let N denote the natural numbers $\{0, 1, 2, \ldots\}$. Let A be an I/O Automaton, and $Q \subseteq \text{states}(A)$ such that all states in Q are reachable. Let I be an arbitrary set, and let $C = \bigcup_{i \in I} C_i$, where each $C_i \in \text{part}(A)$. Let f_A be a function from $\text{states}(A)$ to N. We say that f_A is a *variant function* for (C, Q) if the following conditions hold:

1. For all $(a, \pi, a') \in \text{steps}(A)$ where $a \in Q$:

   (a) If $\pi \in C$, then $f_A(a) = 0$ or $f_A(a) > f_A(a')$.

   (b) If $\pi \notin C$, then $f_A(a) \geq f_A(a')$ and for each $i \in I$, if an action of C_i is enabled from a then an action of C_i is enabled from a'.

   (c) If $f_A(a') > 0$ then $a' \in Q$.

2. For all $a \in Q$, an action of C is enabled from a.

The intuition is that once A enters the set of states Q it may not leave until the value of f_A reaches 0. The actions in C represent those making progress. The following theorem shows that the value of f_A will eventually reach 0 in any fair execution.

Theorem 2.11 *Let A be an I/O Automaton and f_A be a variant function for (C, Q) for some C and Q defined as shown above. Let $x = a_0\pi_1a_1\pi_2\ldots$ be a fair execution of A. Then if $a_i \in Q$ for some i, there exists some $j \geq i$ such that $f_A(a_j) = 0$.*

Proof: Let $f_A(a_i) = n$. If $n > 0$ then we show that there exists some $k > i$ such that $f_A(a_k) < n$ and either $a_k \in Q$ or $f_A(a_k) = 0$. Therefore by induction on n there exists some $j \geq i$ such that $f_A(a_j) = 0$. If $n > 0$, then $a_i \in Q$, and therefore for some $m \in I$, an action of C_m is enabled from a_i. Let $x' = a_i\pi_{i+1}a_{i+i}\ldots$ be the execution fragment of x beginning with state a_i. It cannot be the case that no action of C occurs in x', since fairness to C_m would be violated. Then there exists some $k > i$ such that $\pi_k \in C$. If $f_A(a_{k'}) > 0$ for $i < k' < k$, then all $a_{k'} \in Q$. Therefore $f_A(a_{k-1}) \leq n$, and it follows $f_A(a_k) < n$. □

Note that when showing a function is a variant function we may be able to ignore certain steps if we can show these would not occur, for example due to assumptions on the environment (see Section 2.4). The theorem will still follow restricting x to these executions. A variant function is used in the proof of correctness of a sequential solution to Set Partition (see Theorem 4.3).

2.4 Rely/Guarantee Functions

One way to specify an automaton is to describe its set of fair behaviors. Since automata are input enabled and must respond to any sequence of inputs, some sequences of inputs may cause

an undesired sequence of outputs. In this case we need to show that such sequences of inputs do not occur, and that the automaton will behave correctly for all other input sequences. We can characterize this using *rely/guarantee functions*. A *rely condition* for an automaton A is simply a sequence of input actions of A. For each rely condition there will be a set of behaviors of A which are its "guarantee". The rely/guarantee function for A will be a function mapping each rely condition to the corresponding guarantee.

Let A be an I/O automaton. For an arbitrary set T, let $\mathsf{S}(T)$ be the set of all sequences of elements of T. Let G_A be the function from $\mathsf{S}(\text{in}(A))$ to fbeh(A) such that

$$G_A(r) = \{z : z \in \text{fbeh}(A) \text{ and } z|\text{in}(A) = r\}$$

So essentially G_A partitions fbeh(A) into equivalence classes corresponding to each sequence of input actions r. Each sequence r is a rely condition, whereas the set $G_A(r)$ is the corresponding *guarantee*. The function G_A is the rely/guarantee function for A.

We can extend G_A to take a set of rely conditions as an argument. For $R \subseteq \mathsf{S}(\text{in}(A))$ we can define $G_A(R) = \bigcup_{r \in R} G_A(r)$. It may also be useful to define the following extension for an arbitrary set of schedules R: $G_A(R) = G_A(R|\text{in}(A))$

At first glance it may seem as though there is no real difference between specifying an automaton by its fair behaviors and by specifying it using a rely/guarantee function. However by using rely/guarantee functions one can organize proofs around the assumptions made about the environment and the corresponding guarantees made by each automaton. This organization may make the proof easier to understand. Another advantage is with respect to composition. Although one can derive the fair behaviors of $A \cdot B$ given those for A and B using Theorem 2.4 (see Theorem 2.12), it may still be difficult to characterize exactly what these behaviors are. For example it may be difficult to show that $G_{A \cdot B}(r) \subseteq S$ for some set S. In such a case using successive refinement (Theorem 2.13 below) may simplify the task.

In addition, often one doesn't need to specify $G_A(r)$ precisely. Instead one can specify its essential properties, where a *property* is just a subset of $\mathsf{S}(\text{ext}(A))$. $G_A(r)$ is said to *satisfy* property P if $G_A(r) \subseteq P$.

As an example consider the FIFO (first-in, first-out) buffer $B_{i,j}$ $(i \neq j)$ in Figure 2-1. The buffer takes as input a sequence of messages from a source i and forwards the messages in the same order to a destination j. We can describe a collection of properties $P_{i,j}(r)$ of $B_{i,j}$ as

State Variables:

$buf_{i,j}$: sequence of messages (initially ϵ)

Input Actions:

$send_i(m)$ [for all messages m]

effects:

$buf_{i,j} \leftarrow buf_{i,j} + m$

Output Actions: (each in a separate class of the partition)

$send_j(m)$

preconditions:

$buf_{i,j} \neq \epsilon$

$m = \text{head}(buf_{i,j})$

effects:

$buf \leftarrow \text{tail}(buf_{i,j})$

Figure 2-1: FIFO Buffer $B_{i,j}$

follows. Let $r \in \mathrm{S}(\text{in}(B_{i,j}))$. Let r_j be the sequence resulting from replacing each $send_i(m)$ action of r with $send_j(m)$. Then for $z \in \mathrm{S}(\text{ext}(B_{i,j}))$, $z \in P_{i,j}(r)$ iff

1. $z|\text{out}(B_{i,j}) = r_j$;

2. $r(n)$ precedes $r_j(n)$ in z for all n.

Then it is easily seen that $G_{B_{i,j}}(r)$ satisfies $I_{i,j}(r)$; in fact $G_{B_{i,j}}(r) = P_{i,j}(r)$. Informally, $B_{i,j}$ guarantees to forward all messages sent to it in order, and to not send any other messages.

One can define a rely/guarantee function G_O for any object O. For objects O and P with the same external action signature, O solves P iff $G_O(r) \subseteq G_P(r)$ for all $r \in \mathrm{S}(\text{in}(O))$. Therefore one can specify a problem P as an external schedule module with rely/guarantee function G_P, and show that an object solves P using this technique.

2.4.1 Proof Techniques

We now show how rely/guarantee functions can be used in proofs of correctness. Given rely/guarantee functions for each automaton in a set of automata, we show how to derive the rely/guarantee function for the composition of these automata. We also show how action hiding interacts with rely/guarantee functions.

Composition and Successive Refinement

Theorem 2.12 *Let $\{A_i : i \in I\}$ be a set of strongly compatible objects each with rely/guarantee function G_{A_i}. Let $A = \Pi_i A_i$. Then for $r \in \mathsf{S}(\text{in}(A))$ and $z \in \mathsf{S}(\text{ext}(A))$ where $z|\text{in}(A) = r$, $z \in G_A(r)$ iff $z|A_i \in G_{A_i}(z|\text{in}(A_i))$ for all $i \in I$.*

Proof: Follows immediately from Theorem 2.4. □

For example, consider the composition of $B_{i,j}$ and $B_{i,k}$, where i, j, k are all distinct. Call the composition C; it takes a sequence of messages from source i and forwards duplicate copies to j and k. From Theorem 2.12, it follows $G_C(r) = P_{i,j}(r) \cap P_{i,k}$; in other words for $z \in \mathsf{S}(\text{ext}(C))$ such that $z|\text{in}(C) = r$, $z \in G_C(r)$ iff:

1. $z|\text{out}(B_{i,j}) = r_j$ and $z|\text{out}(B_{i,k}) = r_k$;

2. $r(n)$ precedes $r_j(n)$ and $r_k(n)$ in z for all n.

This is a very simple example since there is no communication between the two automata. For most interesting compositions there is communication among the automata, and it may not be simple to determine the rely/guarantee function for the composition using Theorem 2.12. This is because the rely conditions for each component A_i not only have actions of the environment of A in them, but also actions of the other components of the composition. Furthermore the guarantees of another component A_j may affect the rely conditions of A_i. Therefore while it is simple, for some sequence z, to determine if $z \in G_A(r)$, it may not be so simple to determine if $G_A(r) \subseteq Z$ for some set Z, and this is typically what one is interested in when proving correctness.

Therefore in practice it may be simpler to use the technique of *successive refinement* to describe $G_A(r)$. One defines a sequence of successively smaller sets $Z_0, Z_1, \ldots$ It is shown that the rely condition r from the environment restricts the fair behaviors of A to be in Z_0. From this and the guarantees of the components A_i one can further restrict the fair behaviors to be in set Z_1, and this can be continued step by step. We prove that the guarantees of A will be contained in every Z_j; therefore the Z_j can be thought of as successively refined descriptions of $G_A(r)$.

Theorem 2.13 *Let $\{A_i : i \in I\}$ be a set of strongly compatible objects each with rely/guarantee function G_{A_i}. Let $A = \Pi_i A_i$. Let $r \in \mathsf{S}(\text{in}(A))$, and let $Z_0 = \{z \in \mathsf{S}(\text{ext}(A)) : z|\text{in}(A) = r\}$. Let*

$Z_1, Z_2, \ldots$ *be a sequence of sets such that each satisfies the following* SR (successive refinement) condition*:*

$$Z_{j-1} \supseteq Z_j \supseteq \{z \in Z_{j-1} : z|A_i \in G_{A_i}(z|\text{in}(A_i)) \textit{ for all } i \in I\}$$

Let $Z = \lim_{j\to\infty} Z_j$ *(that is,* Z *is the largest set such that* $Z \subseteq Z_j$ *for all* j*). Then* $G_A(r) \subseteq Z$*.*

Proof: We show for all $z \in G_A(r)$ that $z \in Z$. If $z \in G_A(r)$, then $z|\text{in}(A) = r$, so $z \in Z_0$. Now we show that if $z \in Z_{j-1}$ then $z \in Z_j$. Since $z \in G_A(r)$, it follows $z|A_i \in G_{A_i}(z|\text{in}(A_i))$ for all $i \in I$ from Theorem 2.12. It then follows that if $z \in Z_{j-1}$ it must also be in Z_j from the SR condition. Therefore $z \in Z_j$ for all $j \geq 0$ and it follows $z \in Z$; otherwise $Z \cup \{z\} \subseteq Z_j$ for all j, contradicting the fact that Z is the largest such set. □

Note that the theorem is also easily generalized to a set of rely conditions R (replace r by R in the statement of the theorem). This theorem is similar to to Lemma 3.11 of [Sta84]; however the latter sets up an ordering of the component automata and refines exactly once per automaton.

As an example of successive refinement, consider composing $B'_{i,j}$ and $B_{j,i}$, where $B'_{i,j}$ is a slightly modified version of the buffer having a single message a in its buffer in the initial state. Thus in all fair behaviors $B'_{i,j}$ will send a before any other messages. Call the composition C; notice that C has no input actions and therefore the only rely condition of interest is $r = \epsilon$. For $k \geq 0$, define $p_k = [send_j(a), send_i(a), send_j(a), \ldots]$ (with exactly k elements); so $p_0 = \epsilon$. Then define Z_k to be the set of all sequences $z \in \mathbf{S}(\text{ext}(C))$ such that p_k is a prefix of z. Therefore $Z_0 = \mathbf{S}(\text{ext}(C))$ and the limit Z is the set consisting solely of the infinite alternating sequence $send_j(a), send_i(a), send_j(a), \ldots$. Clearly the SR condition holds for Z_1 since $B'_{i,j}$ guarantees its first output action will be $send_j(a)$ and $B_{j,i}$ guarantees its first output action must follow an input action. For $z \in Z_1$, $[send_j(a)]$ is a prefix of z. From the guarantees of $B_{j,i}$ it follows $[send_i(a)]$ is a prefix of $z|\text{out}(B_{j,i})$ and from the guarantees of $B'_{j,i}$ it follows a second action of $B'_{j,i}$ must follow this $send_i(a)$ action. Therefore $[send_j(a), send_i(a)]$ is a prefix of z, and $z \in Z_2$. In a similar manner it can be shown that the SR condition holds for all Z_k. It follows that $G_C(r) = Z$ and in fact $\text{fbeh}(C) = Z$.

Abstraction

Let s be a set of compatible automata and let S denote the composition of all automata in s. Let sets $s_i : i \in I$ be a partition of s and S_i denote the automaton resulting from the composition of all automata in s_i. Since composition is associative it follows $S = \Pi_i S_i$. However things are more complicated if we are interested in $S' = \text{Hide}_\Sigma(S)$. We cannot simply consider $\text{Hide}_\Sigma(S_i)$ since we may be hiding some actions also in another automaton S_j. The following theorem describes how to hide actions properly in this case.

Theorem 2.14 *Given S and S_i for all $i \in I$ as described above along with Σ, let $S' = \text{Hide}_\Sigma(S)$. Let Ψ be the set of actions contained in the action signatures of two or more automata in $\{S_i\}$ ($\Psi = \bigcup_{i \neq j} \text{acts}(S_i) \cap \text{acts}(S_j)$). For each $i \in I$, let $S'_i = \text{Hide}_{\Sigma-\Psi}(S_i)$ for all $i \in I$. Then*

$$\Pi_i S'_i = \text{Hide}_{\Sigma-\Psi}(S)$$

It follows from this that $\text{Hide}_\Psi(\Pi_i S'_i) = S'$.

Proof: From the definition of action hiding, $S'_i = \text{Hide}_{(\Sigma-\Psi)\cap \text{out}(S_i)}(S_i)$ for all $i \in I$. Furthermore for each i, $[(\Sigma - \Psi) \cap \text{out}(S_i)] \cap \text{acts}(S_j) = \emptyset$ for all $j \neq i$ from the definition of Ψ. It follows from Theorem 2.2 that $\Pi_i S'_i = \text{Hide}_{\Sigma-\Psi}(S)$. □

Furthermore a simple relationship holds between the rely/guarantee function of an automaton and its counterpart with actions hidden.

Theorem 2.15 *Let A be an automaton and Σ be a set of actions. Let $A' = \text{Hide}_\Sigma(A)$. Then for all $r \in \text{S}(\text{in}(A'))$, $G_{A'}(r) = \{z : z = y|\text{ext}(A')$ for some $y \in G_A(r)\}$.*

Proof: Follows immediately from Theorem 2.1. □

These theorems can be used to provide a level of abstraction in correctness proofs. Rather than having to describe the rely/guarantee function for S (which will involve all those actions Σ which are to be hidden and thus could be quite complex) and then derive those for S' from that, one can instead prove correctness of each of the subautomata S'_i, each of which will hide some of the actions, and then combine these results using Theorem 2.14. This will be demonstrated in both the Set Partition and Olympic Torch examples.

Comments

Early works which specified processes in a rely/guarantee style include those by Misra and Chandy ([MC81, MCS82]) and Jones ([Jon83]). The former was used for processes communicating via message passing; the latter for shared variables. The approach here seems closest to that of Stark ([Sta84]) which is not surprising since Stark's model was a precursor to the I/O automaton model. A difference is that Stark allows his rely conditions to be arbitrary predicates over the set of executions, whereas here a set of rely conditions is a predicate of the inputs only. However it seems our rely/guarantee functions are no less expressive than his specifications, particularly since they are simply a partition of the fair behaviors. On the other hand, because of the simple structure of our rely/guarantee functions, we were able to derive a straightforward theorem for their composition (Theorem 2.12) for which there is no analog in Stark's work. He does include a theorem (Lemma 3.11) similar to Theorem 2.13, although it is asymmetric in that an ordering must be defined for the processes and each refinement step corresponds to the rely/guarantee conditions for a single process.

A complication arising from allowing very general rely conditions is that these conditions may not be "realizable"; that is, they may restrict the environment in some way. Abadi and Lamport ([AL90]) give a very general form for specifications using rely/guarantee conditions, but this necessitates defining fairly complicated conditions under which these specifications are realizable. They do no correctness proofs so it is not clear if anything is gained from this generality. On the other hand the restricted form of our rely/guarantee functions assures the behavior of the environment is not restricted.

Chapter 3

Process Creation

A way to handle process creation in a static model is to start with the set of all possible processes, and define a predicate for each process which indicates whether or not it is alive. Then one must show the process does not actually do anything when it is not alive. This is shown below for the I/O automata model.

3.1 Definitions

We can model a system in which processes are created and destroyed as the set of all processes (where each process is an I/O automaton) that could possibly be alive during any execution of the system. The number of such processes could be infinite, but will typically be at most countably infinite. For each I/O automaton A, define a boolean function $w_A()$ whose domain is states(A). We say that A is *alive* at state a if $w_A(a) = \mathit{true}$; it is *dead* at a otherwise. Typically a component of a will be a boolean variable named *working*, in which case we can set $w_A(a) = a.\mathit{working}$.

In addition all automata in such a system are required to satisfy properties that correspond to the intuitive notion of process creation. For all $(a, \pi, a') \in$ steps(A) we require the following:

1. If π is an output or internal action, then $w_A(a) = \mathit{true}$.

2. If π is an input action, then if $w_A(a) = w_A(a') = \mathit{false}$, it must be the case that $a = a'$. If $w_A(a) = \mathit{false}$ and $w_A(a') = \mathit{true}$ we call π a *creation action.*

This says only a live automaton may take steps; however, some input actions may be creation actions. If π is a creation action, we normally would like to have the additional requirement that for all $(b, \pi, b') \in$ steps(A), if $w_A(b) = true$ then $b' = b$. This says that π has no effect if the process is already alive. However we would like to compose automata and define a function w for the composition. Thus if A is the composition of $\{A_i : i \in I\}$ where w_{A_i} is defined for all $i \in I$, then define $w_A(a) = true$ iff there exists some i for which $w_{A_i}(a|A_i) = true$. This says that A is alive iff one of its components is. Therefore the composition A may be alive and yet additional creation actions will have effect, as they create other components of A. We define $w_{A_i}(a) = w_{A_i}(a|A_i)$.

A *dynamic system* is a collection of automata $\{A_i : i \in I\}$ along with, for each automaton A_i, a function w_{A_i} satisfying the conditions above.

In order to make for more exciting reading the following terms will be occasionally be used. Let A be a dynamic system that is the composition of processes $\{A_i : i \in I\}$ and let $x \in$ execs(A) where $x = a_0\pi_1a_1\pi_2a_2\ldots$. Then A_i is said to be *alive* at a_j if $w_{A_i}(a_j) = true$ and *dead* at a_j otherwise. It *dies* at a_j if it is dead at a_j and alive at a_{j-1}; it is *created* at a_j if it is alive at a_j and dead at a_{j-1}. Thus in the latter case π_j would necessarily be a creation action. If $w_{A_i}(a_0) = true$ then we say A_i is *created* at a_0. Note that if we want to consider A itself (even if it is not a composition of other automata) we can use the above with w_A in place of w_{A_i} provided w_A has been defined.

Process A is said to *create* B in execution x if there is an output action of A in x that creates B. Process A is called the *parent* of B and B a *child* of A. A *descendant* of A is either a child of A or a child of a descendant of A. The *process creation tree* of A is A along with all its descendants.

3.2 Abstraction and Rely/Guarantee Functions

Specifications using rely/guarantee functions are particularly useful in dynamic systems. When a process creates children, we can think of these children as satisfying some of the guarantees of the parent, given that the parent and the overall environment satisfy certain rely conditions. Therefore it is useful to define a rely/guarantee function for the entire process creation tree of A. If the composition of the processes in this tree is C, then the specification of C can be derived

from the specifications of A and the process creation trees of its children using Theorem 2.12 or Theorem 2.13. However typically one will wish to hide internal actions of the tree, in this case one also uses Theorem 2.14 and Theorem 2.15. These techniques will be demonstrated in both the Set Partition and Olympic Torch examples.

3.3 Comments

Although it may seem at first that reasoning about a dynamic system is very complicated, it can be seen that it need not be. Although we consider an infinite set of automata, only those which are alive are of concern. The dynamic topology can be handled well by using rely/guarantee functions; a process can rely on messages arriving even if it cannot tell who will send the message.

Because process creation is such a general phenomenon, it is difficult to prove any useful theorems for it without restricting the class of examples. However properties of process creation will be used throughout the proofs of correctness for Set Partition and Olympic Torch. An example of such a property is that if a process is initially dead, it cannot send any message until a create action occurs. This is used to prove safety properties.

Chapter 4

Set Partition

4.1 Specification

The Set Partition problem is to start with two nonempty disjoint finite sets of integers S and L, and to return two of integers S' and L' such that

1. $S' \cup L' = S \cup L$

2. $|S'| = |S|$ and $|L'| = |L|$

3. $\max(S') \leq \min(L')$

In other words, all the smaller integers will be in S' and the larger integers in L' at the end. This problem is used as an example in [Bar85] to compare several verification methods for parallel programs; a comparison with this method can be found in Section 4.4. In this chapter three different solutions to Set Partition are presented in increasing order of complexity: a nondeterministic algorithm which guesses the solution; a deterministic sequential algorithm which swaps elements of S and L; and a distributed solution in which the swapping is done asynchronously.

We can specify Set Partition by an external schedule module $\mathcal{Z}$ whose action signature is in $= \{start(S, L)\}$ and out $= \{stop(S, L)\}$ for all possible pairs (S, L) of nonempty disjoint finite sets of integers. We will specify the behaviors of $\mathcal{Z}$ using a rely/guarantee function (see Section 2.4). We will only be concerned with the case in which the environment sends a single

State Variables:

S, L: *sets of integers* (initially $\emptyset$)

Input Actions:

start(S',L')

effects:

if $S = \emptyset$ and S' and L' are nonempty disjoint finite sets of integers then

$S \leftarrow S'$

$L \leftarrow L'$

Output Actions:

stop(S',L')

preconditions:

$S \neq \emptyset$

$S' \cup L' = S \cup L$

$|S'| = |S|$ and $|L'| = |L|$

$\max(S') \leq \min(L')$

effects:

$S \leftarrow \emptyset$

Figure 4-1: States and actions of $\mathcal{P}_1$.

start(S, L) action. Therefore if $r_{S,L} = [start(S, L)]$ where S and L are nonempty disjoint finite sets of integers, then $G_{\mathcal{Z}}(r_{S,L}) = \{[start(S, L), stop(S', L')]\}$ where (S, L) and (S', L') satisfy the three conditions above. For all other $r \in \mathsf{S}(\text{in}(\mathcal{Z}))$, define $G_{\mathcal{Z}}(r) = \mathsf{S}(\text{ext}(\mathcal{Z}))$; in other words, we do not care what the behavior is in this case.

The following theorem should be obvious.

Theorem 4.1 *Given nonempty disjoint finite sets of integers S and L, S' and L' satisfying the three conditions above exist and are unique.*

A simple nondeterministic solution to Set Partition is given by $\mathcal{P}_1$ shown in Figure 4-1. This automaton is completely straightforward: it receives the two sets S and L and returns S' and L' satisfying the above conditions. The state is characterized by the variables S and L; notice that S is being used to determine whether the process is alive or not; thus $w_{\mathcal{P}_1}(a) = (a.S \neq \emptyset)$ (see Section 3).

We now want to show that $\mathcal{P}_1$ solves $\mathcal{Z}$, which is straightforward.

Theorem 4.2 $\mathcal{P}_1$ *solves* $\mathcal{Z}$.

State Variables:

S, L: *sets of integers* (initially $\emptyset$)

Input Actions:

start(S',L')

effects:

if $S = \emptyset$ and S' and L' are nonempty disjoint finite sets of integers then

$S \leftarrow S'$

$L \leftarrow L'$

Output Actions: (all local actions are in one partition)

stop(S,L)

preconditions:

$S \neq \emptyset$

$\max(S) \leq \min(L)$

effects:

$S \leftarrow \emptyset$

Internal Actions:

compute

preconditions:

$S \neq \emptyset$

$\max(S) > \min(L)$

effects:

$S \leftarrow S + \min(L) - \max(S)$

$L \leftarrow L + \max(S) - \min(L)$

Figure 4-2: States and actions of $\mathcal{P}_2$.

Proof: Let $r_{S,L} = [start(S, L)]$ for some pair of nonempty disjoint finite sets of integers S and L. We need to show that $G_{\mathcal{P}_1}(r_{S,L}) \subseteq G_{\mathcal{Z}}(r_{S,L})$. Let $x = a_0\pi_1a_1\ldots$ be a fair execution of $\mathcal{P}_1$ such that $z = \text{sched}(x)$ and $z|\text{in}(\mathcal{P}_1) = r_{S,L}$. Since no action of out($\mathcal{P}_1$) is enabled from a_0, it must be the case that $\pi_1 = start(S, L)$. By Theorem 4.1 there exists exactly one pair (S', L') such that $stop(S', L')$ is enabled in a_1. Therefore by fairness we must have $\pi_2 = stop(S', L')$. Due to the conditions on *stop*, (S, L) and (S', L') will satisfy the conditions of Set Partition. In state a_2, no output action of $\mathcal{P}_1$ will be enabled, and since by hypothesis no input action will occur, $z = [start(S, L), stop(S', L')]$, and thus $z \in G_{\mathcal{Z}}(r_{S,L})$. □

4.2 Deterministic Sequential Implementation

A simple sequential algorithm to solve Set Partition swaps $\max(S)$ and $\min(L)$ until con-

dition 3 is satisfied. We implement this algorithm using I/O Automaton $\mathcal{P}_2$. See Figure 4-2. The process is similar to $\mathcal{P}_1$ save that we have added an internal action to swap $\max(S)$ and $\min(L)$ and have changed the preconditions of the output actions. The allowed states and external action signature of the two automata are identical, and we define $w_{\mathcal{P}_2}(a) = (a.S \neq \emptyset)$ as before. There is only one partition containing all local (output and internal) actions. When $w_{\mathcal{P}_2}(a) = \mathit{true}$, exactly one of the *compute* and *stop* actions will be enabled, and since it will be continuously enabled by fairness it will occur. Note that the two clauses in *effects* of the *compute* actions are meant to take place simultaneously, thus swapping $\max(S)$ and $\min(L)$ in each set as desired.

We wish to show that $\mathcal{P}_2$ solves $\mathcal{Z}$. The proof is similar to Theorem 4.2 save that we now have to use an invariant and a variant function.

Theorem 4.3 *$\mathcal{P}_2$ solves $\mathcal{Z}$.*

Proof: Let $r_{S,L} = [\mathit{start}(S,L)]$ for some pair of nonempty disjoint finite sets of integers S and L. We need to show that $G_{\mathcal{P}_2}(r_{S,L}) \subseteq G_{\mathcal{Z}}(r_{S,L})$. Let $x = a_0\pi_1 a_1 \ldots$ be a fair execution of $\mathcal{P}_2$ such that $z = \mathrm{sched}(x)$ and $z|\mathrm{in}(\mathcal{P}_2) = r_{S,L}$. Since no action of $\mathrm{out}(\mathcal{P}_2)$ is enabled from a_0, it must be the case that $\pi_1 = \mathit{start}(S, L)$.

Define a boolean function $P_x(a) = [(a_1.S \cup a_1.L = a.S \cup a.L) \wedge (|a_1.S| = |a.S|) \wedge (|a_1.L| = |a.L|)]$. In other words, P is the first two conditions of Set Partition, and is an invariant that captures the safety properties. Let $x_i = a_0\pi_1 \ldots a_i$. We want to show that, for all $i \geq 1$, if no *stop* action occurs in x_i then $P_x(a_i) = \mathit{true}$. This is done by induction; clearly it holds for $i = 1$. Now assume it holds for x_{i-1}; prove for $i > 1$. Consider the possible actions π_i. By hypothesis we cannot have $\pi_i = \mathit{start}(S', L')$. If $\pi_i = \mathit{compute}$, then from the effects and the induction hypothesis it can easily be seen that $P_x(a_i) = \mathit{true}$; note that this is dependent upon S and L being disjoint.

We now must show that there exists some $i > 1$ such that $\pi_i = \mathit{stop}(a_{i-1}.S, a_{i-1}.L)$ (and for no $i' < i$ is $\pi_{i'}$ a *stop* action). Since the precondition of π_i is that $\max(a_{i-1}.S) \leq \min(a_{i-1}.L)$ and since $P_x(a_{i-1})$ holds, it must be the case that $(a_1.S, a_1.L)$ and $(a_{i-1}.S, a_{i-1}.L)$ satisfy all three Set Partition conditions. Furthermore since the effect of this action is to set $a_i.S = \emptyset$, no local action of $\mathcal{P}_2$ will be enabled from a_i, and a *start* action cannot occur by hypothesis. Since π_1 and π_i are the only external actions in $z = \mathrm{sched}(x)$, it must be the case that $z \in G_{\mathcal{Z}}(r_{S,L})$.

Let $C = \text{local}(\mathcal{P})$ and let $Q = \{a \in \text{states}(\mathcal{P}_2) : |a.S| > 0\}$. Define variant function $f(a) = |a.\hat{S}|$, where $a.\hat{S} = \{s \in a.S : s > \min(a.L)\}$. We show that f is a variant function for (C, Q). When $a \in Q$, either a *stop* or *compute* action will be enabled from a, so condition 2 holds. For condition 1, let $(a, \pi, a') \in \text{steps}(A)$ where $a \in Q$. If $\pi = start(S', L')$, then since $|a.S| > 0$ we must have $a' = a$. If $\pi = stop(a.S, a.L)$, then it must be the case that $f(a) = 0$ from the preconditions. Finally if $\pi = compute$, then by the preconditions we must have $f(a) > 0$. Let $S_1 = a.\hat{S}$ and $S_2 = a'.\hat{S}$. Since $|S_1| > 0$, $\max(a.S) \in S_1$. Furthermore since $\min(a'.L) \geq \min(a.L)$ it follows that $\min(a.L) \notin S_2$ and also for all $s \in a.S \cap a'.S$, if $s \notin S_1$ then $s \notin S_2$. Thus from the effects of *compute*, $|S_2| < |S_1|$. Thus f is a variant function for (C, Q) and it follows from Theorem 2.11 that for any fair execution x of $\mathcal{P}_2$, since $a_1 \in Q$ there must be some $j > 1$ such that $f(a_j) = 0$. If $a_j.S = \emptyset$, then a *stop* action must have occurred in x_j. Otherwise the only local action enabled from a_j is $stop(a_j.S, a_j.L)$; it is continuously enabled (despite any occurrences of input actions) and thus must occur due to fairness. □

4.3 Distributed Solution with Instantaneous Message Passing

We implement Set Partition in a distributed system in which message transmission time is instantaneous. Process creation occurs in this version, but it is very simple. One would not expect the process creation to be troublesome, and in fact it isn't.

See Figures 4-3, 4-4 and 4-5 for the states and actions of the three automata used. The general idea is that $\mathcal{P}_3$ creates two processes $\mathcal{S}_3$ and $\mathcal{L}_3$, passing them sets S and L respectively. The latter two processes then swap elements as in the sequential algorithm and return the answer to $\mathcal{P}_3$. The proof of correctness will be similar to that for the sequential solution, but complications will arise from having multiple processes. Process creation can be handled very simply and will not cause problems. Define $w_{\mathcal{P}_3}(a) = a.working$, $w_{\mathcal{S}_3}(a) = (S \neq \emptyset)$ and $w_{\mathcal{L}_3}(a) = (L \neq \emptyset)$. It's easy to see from the code that the set of these three processes is a dynamic system. We don't derive the variable *working* of $\mathcal{P}_3$ from S and L because we use those variables in another way: When $\mathcal{P}_3$ suspends itself waiting for answers from $\mathcal{S}_3$ and $\mathcal{L}_3$, it sets S and L to each be $\emptyset$. When the latter processes return answers the variables will be instantiated with the new sets.

State Variables:

working: *boolean* (initially *true*)

S, L: *sets of integers* (initially $\emptyset$)

Input Actions:

start(S',L')

effects:

if *working* = *false* and S' and L' are nonempty disjoint finite sets of integers then

$S \leftarrow S'$

$L \leftarrow L'$

working $\leftarrow$ *true*

Sreturn(S')

effects:

if *working* = *true* and $S = \emptyset$

$S \leftarrow S'$

Lreturn(L')

effects:

if *working* = *true* and $L = \emptyset$

$L \leftarrow L'$

Output Actions: (all in one class)

stop(S,L)

preconditions:

working = *true*

$S \neq \emptyset$ and $L \neq \emptyset$

$\max(S) \leq \min(L)$

effects:

working $\leftarrow$ *false*

call(S,L)

preconditions:

working = *true*

$S \neq \emptyset$ and $L \neq \emptyset$

$\max(S) > \min(L)$

effects:

$S \leftarrow \emptyset$

$L \leftarrow \emptyset$

Figure 4-3: States and Actions of $\mathcal{P}_3$

State Variables:

   S: *set of integers* (initially $\emptyset$)

   buf: *sequence of integers* (initially ϵ)

Input Actions:

   call(S',L')

      effects:

         if $S = \emptyset$ and S' is a nonempty sets of integers then

            $S \leftarrow S'$

            $buf \leftarrow [\min(L')]$

   Lsend(n)

      effects:

         if $S \neq \emptyset$

            $buf \leftarrow buf + n$

Output Actions: (all in one class)

   Sreturn(S)

      preconditions:

         $S \neq \emptyset$

         $buf \neq \epsilon$ and $\max(S) \leq \text{head}(buf)$

      effects:

         $S \leftarrow \emptyset$

   Ssend(n)

      preconditions:

         $S \neq \emptyset$

         $buf \neq \epsilon$ and $\max(S) > \text{head}(buf)$)

         $n = \max(S - \max(S) + \text{head}(buf))$

      effects:

         $S \leftarrow S - \max(S) + \text{head}(buf)$

         $buf \leftarrow \text{tail}(buf)$

Figure 4-4: States and Actions of S_3

(Note that the code is identical to $\mathcal{S}_3$ save that S is replaced by L, $\leq$ by $\geq$, $>$ by $<$, and max and min are switched throughout.)

State Variables:

L: *set of integers* (initially $\emptyset$)

buf: *sequence of integers* (initially ϵ)

Input Actions:

call(S',L')

effects:

if $L = \emptyset$ and L' is a nonempty sets of integers then

$L \leftarrow L'$

$buf \leftarrow [\max(S')]$

Ssend(n)

effects:

if $L \neq \emptyset$

$buf \leftarrow buf + n$

Output Actions: (all in one class)

Lreturn(L)

preconditions:

$L \neq \emptyset$

$buf \neq \epsilon$ and $\min(L) \geq \text{head}(buf)$

effects:

$L \leftarrow \emptyset$

Lsend(n)

preconditions:

$L \neq \emptyset$

$buf \neq \epsilon$ and $\min(L) < \text{head}(buf)$

$n = \min(L - \min(L) + \text{head}(buf))$

effects:

$L \leftarrow L - \min(L) + \text{head}(buf)$

$buf \leftarrow \text{tail}(buf)$

Figure 4-5: States and Actions of $\mathcal{L}_3$

The processes $\mathcal{S}_3$ and $\mathcal{L}_3$, in addition to maintaining the set they are responsible for, also have a buffer to store incoming messages from the other process. When one arrives it is automatically stored in the buffer. Due to the fact that processes buffer messages instead of an autonomous mail system, we must create both $\mathcal{S}_3$ and $\mathcal{L}_3$ at the same time if we wish them to be symmetric. Otherwise if one is created first it could send a message to the other which would be lost. This problem could be easily overcome by having the processes send *ready* messages to each other both when created and when receiving a *ready* message from the other process. Each process could then send messages normally once it receives a *ready* message from the other process.

Each process is to send an initial message, then receive a message and send the next message, and so forth. The initial message is different from the others in that no message need be received before it is can be sent. The convention used in [Bar85] is to put ∞ or $-\infty$ in the input buffer initially; in the algorithm here we put $\min(L)$ in the buffer of $\mathcal{S}_3$ and $\max(S)$ in the buffer of $\mathcal{L}_3$ initially; this cleans up the code and the proof considerably by avoiding special cases. Notice that the *send* actions of the I/O Automata remove a message from the input buffer (which is essentially a *receive*) and send a message simultaneously; in other implementations these would be done as separate actions. However it is not hard to see that even if they were implemented as separate actions correctness would be maintained.

We compose our three automata (which are strongly compatible) to produce a single automaton $\mathcal{N}_3$. First let $\mathcal{N}_3' = \mathcal{P}_3 \cdot \mathcal{S}_3 \cdot \mathcal{L}_3$. Let Σ be the set of all actions of $\mathcal{N}_3'$ save the *start* and *stop* actions. Then let $\mathcal{N}_3 = \text{Hide}_\Sigma(\mathcal{N}_3')$ Thus $\mathcal{N}_3$ will have the same external action signature as $\mathcal{Z}$. Note that the partition of $\mathcal{N}_3$ has three classes, one for each of the automata in the composition. Since we have used the same names for the state variables of the components, we need to distinguish them in the composition. We do this by subscripting the variable name with the automaton name. For example, $buf_\mathcal{S}$ is the *buf* state variable of process $\mathcal{S}_3$.

One could use invariants and a variant function as in the proof of Theorem 4.3, but these are much more complicated in this case. Therefore we will instead use rely/guarantee functions and successive refinement (see Section 2.4). This method too is somewhat complicated here, but still much simpler than the traditional proof methods. Furthermore, we can break the proof into parts. Since the role of $\mathcal{P}_3$ is simply to start the parallel computation and collect

the results, it makes sense to consider it separately. The computation takes place between $\mathcal{S}_3$ and $\mathcal{L}_3$, and it simplifies matters to hide this from $\mathcal{P}_3$. Therefore let $C' = \mathcal{S}_3 \cdot \mathcal{L}_3$, and let $C = Hide_\Gamma(C')$, where Γ is the set of all *Ssend* and *Lsend* actions. Let Ψ be the set of all *call*, *Sreturn* and *Lreturn* actions. It then follows that $\mathcal{N}_3 = \text{Hide}_\Psi(P)$ from Theorem 2.14.

We will therefore first specify C, and then show how to combine this specification with a specification of $\mathcal{P}_3$ to produce a specification for $\mathcal{N}_3$, and finally show that this solves $\mathcal{Z}$.

Lemma 4.4 *If* $r = [\text{call}(S, L)]$ *for nonempty disjoint finite sets of integers* S *and* L, *then*

$$G_C(r) = \{[\text{call}(S,L), \text{Sreturn}(S'), \text{Lreturn}(L')], [\text{call}(S,L), \text{Lreturn}(L'), \text{Sreturn}(S')]\}$$

where S' *and* L' *satisfy the conditions of Set Partition. Furthermore if* r *is a prefix of* $q \in \mathsf{S}(\text{in}(C))$ *(note that* q *must be a sequence of* call *actions), then one of the two sequences in* $G_C(r)$ *will be a prefix of any sequence in* $G_C(q)$. *If* $q = \epsilon$ *then* $G_C(q) = \epsilon$.

Proof: As the distributed algorithm is similar in concept to the sequential algorithm, it is useful to structure the proof in a similar manner as well. Fix S and L to be nonempty disjoint finite sets of integers, and define $S_0 = S$ and $L_0 = L$. Define $S_i = S_{i-1} - s_{i-1} + l_{i-1}$ and $L_i = L_{i-1} - l_{l-1} + s_{i-1}$ if $s_{i-1} > l_{i-1}$; $S_i = S_{i-1}$ and $L_i = L_{i-1}$ otherwise, where $s_i = \max(S_i)$ and $l_i = \min(L_i)$. Notice that $S_i \cup L_i = S \cup L$, $|S_i| = |S|$ and $|L_i| = |L|$ for all i. Furthermore there exists some j such that $s_j \leq l_j$. Define $\hat{S}_i = \{s \in S_i : s > \min(L_i)\}$ and then define $f(i) = |\hat{S}_i|$. It can easily be seen that $f(i) > f(i')$ if $f(i) > 0$ and $i < i'$, and that $f(j) = 0$ iff $s_j \leq l_j$. Define t to be the smallest number j such that $s_j \leq l_j$.

This was the essence of the proof of Theorem 4.3, and will be the essence of this proof as well. The notation above will be used throughout the proof.

We will first specify $\mathcal{S}_3$ and $\mathcal{L}_3$, and then combine the specifications using successive refinement to determine the specification of the composition C'. It will then be shown that after hiding the actions in Γ the above specification of C is met.

First we specify $\mathcal{S}_3$; this will be done by specifying properties of $G_{\mathcal{S}_3}(r)$ for those r of interest. First note that for any $r \in \mathsf{S}(\text{in}(\mathcal{S}_3))$, if there is no *call* action in r then $G_{\mathcal{S}_3}(r) = r$; in other words $\mathcal{S}_3$ does nothing. The *call* action is a "create" action for $\mathcal{S}_3$ so since the process is initially dead it cannot do anything until it is created. Furthermore in any r for which there is a *call* action there can be no local action of $\mathcal{S}_3$ before the first *call* action in any $z \in G_{\mathcal{S}_3}(r)$.

Foir all $i \geq 0$, define

$$g_i^s = [Ssend(s_1), Ssend(s_2), \ldots, Ssend(s_i)]$$
$$g_i^l = [Lsend(l_1), Lsend(l_2), \ldots, Lsend(l_i)]$$

It then follows that if $[call(S, L)] \cdot g_i^l$ is a prefix of r and $i < t$, then g_{i+1}^s is a prefix of $r|\text{out}(\mathcal{S}_3)$ and $Ssend(s_j)$ follows $Lsend(l_{j-1})$ for all $j > 1$; also $Ssend(s_1)$ follows $call(S, L)$. This is easily checked by examining the code of $\mathcal{S}_3$.

On the other hand if $[call(S, L)] \cdot g_t^l$ is a prefix of r and there are no additional *call* actions in r, then

$$r|\text{out}(\mathcal{S}_3) = g_i^s \cdot [Sreturn(S_i)]$$

and $Sreturn(S_i)$ follows $Lsend(l_i)$. If there are additional *call* messages after $call(S, L)$ then this sequence is at least a prefix of $r|\text{out}(\mathcal{S}_3)$. Again this is all easily checked by examining the code.

The specification of $\mathcal{L}_3$ is analogous. If $[call(S, L)] \cdot g_i^s$ is a prefix of r and $i < t$, then g_{i+1}^l is a prefix of $r|\text{out}(\mathcal{L}_3)$ and $Lsend(l_j)$ follows $Ssend(s_{j-1})$ for all $j > 1$; also $Lsend(l_1)$ follows $call(S, L)$. If $[call(S, L)] \cdot g_t^s$ is a prefix of r and there are no additional *call* actions in r, then $r|\text{out}(\mathcal{L}_3) = g_i^l \cdot [Lreturn(L_i)]$ and $Lreturn(L_i)$ follows $Ssend(s_i)$. If there are additional *call* messages after $call(S, L)$ then this sequence is at least a prefix of $r|\text{out}(\mathcal{L}_3)$. As with $\mathcal{S}_3$ this is easily checked by examining the code.

We combine the specifications of $\mathcal{S}_3$ and $\mathcal{L}_3$ to derive a specification of C'. This is done using successive refinement. In order to prove the theorem we have to consider three cases (where $r \in \mathrm{S}(\text{in}(C'))$):

1. $r = \epsilon$;
2. $r = [call(S, L)]$;
3. $[call(S, L)]$ is a prefix of r.

In the first case it is trivial to show that $G_{C'}(r) = \epsilon$, since both $\mathcal{S}_3$ and $\mathcal{L}_3$ guarantee to do nothing. We will prove the second case here; the proof of the third case is handled in an analogous manner. Therefore set $r = [call(S, L)]$. Z_0 is defined to be the set of $z \in \mathrm{S}(\text{ext}(C')$ such that $z|\text{in}(C') = r$.

The definition of Z_i will depend upon whether or not $i < 2t$. Intuitively Z_i for $i < 2t$ includes sequences representing computation that has not yet been completed, while Z_i for $i \geq 2t$ only includes sequences in which the computation has completed.

Let $y_{i,j}$, where $|i-j| \leq 1$, be the set of all sequences $y \in \mathrm{S}(\mathrm{ext}(C'))$ such that $[call(S,L)] \cdot g^l_i$ is a prefix of $y|\mathrm{in}(\mathcal{S}_3)$ and $[call(S,L)] \cdot g^s_i$ is a prefix of $y|\mathrm{in}(\mathcal{L}_3)$, and also $Ssend(s_k)$ follows $Lsend(l_{k-1})$ (for all $k \leq i$) and $Lsend(l_k)$ follows $Ssend(s_{k-1})$ (for all $k \leq j$) in y. For $k < 2t$, define Z_k to be the union of all $y_{i,j}$ in which $i + j = k$ and $|i-j| \leq 1$. Note that the condition $|i-j| \leq 1$ will imply, once the proof is completed, that $buf_{\mathcal{S}}$ and $buf_{\mathcal{L}}$ will each need to be of size at most two.

Define Z_k for $k \geq 2t$ to be the set of all $y \in y_{t,t}$ such that

$$y|\mathrm{out}(\mathcal{S}_3) = g^s_t \cdot [Sreturn(S_t)] \text{ and}$$
$$y|\mathrm{out}(\mathcal{L}_3) = g^l_t \cdot [Lreturn(L_t)],$$

and also $Sreturn(S_t)$ follows $Lsend(l_t)$ and $Lreturn(L_t)$ follows $Ssend(s_t)$.

We show for all $k \geq 0$ that if Z_k satisfies the SR condition then so does Z_{k+1}. First consider the case $k < 2t-1$. Let $z \in Z_k$, so $z \in y_{i,j}$ for some (i,j) such that $i + j = k$ and $|i-j| \leq 1$. Therefore $i, j < t$, so $s_m \leq l_m$ for all $m \leq \max(i,j)$. Therefore from the properties of the guarantees of $\mathcal{S}_3$ and $\mathcal{L}_3$ it follows $z \in y_{i+1,j} \cup y_{i,j+1}$ if $i = j$; $z \in y_{i+1,j}$ if $i < j$; and $z \in y_{i,j+1}$ otherwise. In all three cases it follows $z \in Z_{k+1}$.

If $k = 2t-1$, then if $z \in Z_k$ it follows $z \in y_{i,j}$ where either $i = t$ or $j = t$. Say that $i = t$ and $j = t-1$; the other case is analogous. From the properties of the guarantees of $\mathcal{S}_3$, it follows $z|\mathrm{out}(\mathcal{S}_3) = g^s_t \cdot [Sreturn(S_t)]$ and $Sreturn(S_t)$ follows $Lsend(l_t)$. Then from the properties of $\mathcal{L}_3$ it follows $z|\mathrm{out}(\mathcal{L}_3) = g^l_t \cdot [Lreturn(L_t)]$ and $Lreturn(L_t)$ follows $Ssend(s_t)$. Therefore $z \in Z_{k+1}$.

The case of $k \geq 2t$ is trivial since $Z_{k+1} = Z_k$. Therefore $G_{C'}(r) \subseteq Z_{2t}$. The theorem follows upon hiding the ***Ssend*** and ***Lsend*** actions and applying Theorem 2.15. □

Theorem 4.5 *$\mathcal{N}_3$ solves $\mathcal{Z}$.*

Proof: Let $r_{S,L} = [start(S,L)]$ for some pair of nonempty disjoint finite sets of integers S and L. We need to show that $G_{\mathcal{N}_3}(r_{S,L}) \subseteq G_{\mathcal{Z}}(r_{S,L})$. Let $x = a_0\pi_1 a_1 \ldots$ be a fair execution of $\mathcal{N}_3$ such that $z = \mathrm{sched}(x)$ and $z|\mathrm{in}(\mathcal{N}_3) = r_{S,L}$. Since no action of $\mathrm{out}(\mathcal{N}_3)$ is enabled from a_0, it must be the case that $\pi_1 = start(S,L)$.

First note that if $\max(S) \leq \min(L)$, it is trivial to show that $G_{\mathcal{N}_3}(r_{S,L}) \subsetneq G_{\mathcal{Z}}(r_{S,L})$. Therefore in what follows assume $\max(S) > \min(L)$.

Let $P = \mathcal{P}_3 \cdot C$. The rely/guarantee functions of $\mathcal{P}_3$ and C are simple enough that we can determine the rely/guarantee function of P from them using Theorem 2.12. For $z \in \mathbf{S}(\text{ext}(P))$ such that $z|\text{in}(P) = r_{S,L}$, we have already noted that $[\textit{start}(S,L)]$ must be the first action of z. Since from the guarantees of C it can be deduced that any external action of C must follow an *call* action, and since we are assuming $\max(S) > \min(L)$, it follows that $\textit{call}(S,L)$ must be the next action of Z. From the guarantees of $\mathcal{P}_3$ it can be deduced that there will be no output action of $\mathcal{P}_3$ until both *Sreturn* and *Lreturn* actions occur; from the guarantees of C it follows then that the next two actions of z are $\textit{Sreturn}(S')$ and $\textit{Lreturn}(L')$ (in either order), where S' and L' satisfy the sift conditions. From the guarantees of $\mathcal{P}_3$ it follows the next action will be $\textit{stop}(S', L')$, and from the guarantees of both it follows there will be no other actions.

In summary:

$$\begin{aligned} G_P(r_{S,L}) &= \{[\textit{start}(S,L), \textit{call}(S,L), \textit{Sreturn}(S'), \textit{Lreturn}(L'), \textit{stop}(S',L')], \\ &\quad [\textit{start}(S,L), \textit{call}(S,L), \textit{Lreturn}(L'), \textit{Sreturn}(S'), \textit{stop}(S',L')]\} \end{aligned}$$

Now if Ψ is the set of all *call*, *Sreturn* and *Lreturn* actions, it follows that $\mathcal{N}_3 = \text{Hide}_\Psi(P)$ from Theorem 2.14. Then from Theorem 2.15 it follows

$$G_{\mathcal{N}_3}(r_{S,L}) = \{[\textit{start}(S,L), \textit{stop}(S',L')]\}$$

and the theorem is proven. □

4.4 Comments and Comparisons

Barringer ([Bar85]) uses the Set Partition example for two demonstrations of proof methods in his survey: the method of Owicki and Gries for parallel programs that communicate via shared variables ([OG76]) and the method of Levin and Gries for correctness of Communicating Sequential Processes ([LG81]). It is not really fair to compare this proof to either of them directly, as the proof methods are based upon different models. The proof using Owicki and Gries method is very complicated, but this is partly due to the nature of shared variable

algorithms. The proof using Levin and Gries uses synchronized message passing and is much simpler.

What can be compared, though, is the overall style. The proofs using both methods are essentially proofs of invariants (only safety properties were proved by Barringer) and are similar in this way to the invariant proof in Theorem 4.3, although more complicated due to the parallelism of the algorithm. The problem with an invariant is that one must consider all possible states of the system, and there can be many in a system with multiple processes. So there can be no modularity, and the invariant itself can be quite difficult to understand.

On the other hand the proof using rely/guarantee functions seems to follow the structure of the algorithm, particularly the sequence of communications, more closely. Also we were able to break the problem into parts, considering $\mathcal{S}_3$ and $\mathcal{L}_3$ separately. For these reasons the proof seems considerably simpler. An additional advantage is that liveness is also handled by rely/guarantee functions.

The proof of Set Partition in [dC89] uses a sort of rely/guarantee approach, but manages to be complicated despite this. Furthermore, even disregarding the fact that there is no underlying model, it is not at all clear that correctness has been proved. De Champeaux shows that rely/guarantee conditions for his processes are "compatible," meaning that the correct behaviors of the system are also behaviors of each process. But this is just half the proof. One also needs to show that the correct behaviors are the only possible behaviors for the system, something that is not done in his proof.

Chapter 5

Olympic Torch

5.1 Description of Olympic Torch

Olympic Torch is the name given by Dennis DeChampeaux to a kind of divide-and-conquer approach using process creation. It was inspired by an example given by Ehud Shapiro in a class on concurrent Prolog. If we need to perform several tasks whose outputs must be ordered, we can do this using a single process, or the process can create a number of children each of which will perform a subset of the tasks. The output ordering is preserved by the children sending each other messages, indicating when they are done and when the next process may output. Of course these children can themselves create children, and the complications arising from asynchronous process creation and message routing are what make the Olympic Torch interesting. If we consider the process creation tree, no computation or message passing is done save at the leaves of the tree. The synchronization can then be seen as a flow of messages through the leaves indicating when the next process can output. This could be viewed as the Olympic Torch being passed from runner to runner, thus the name.

To be more concrete, say that the input is a finite sequence $s = [s_1, s_2, \ldots, s_q]$ and for some function f we want to print the sequence $[f(s_1), f(s_2), \ldots, f(s_q)]$. (The input could also be a binary tree whose leaves when ordered form the sequence s.) We could do this using a sequential process that computes $f(s_i)$ and sends the results in order, but if f is a time-consuming function we may want to compute it in a distributed manner. Thus we create two processes, sending the first half of s to the left child and the second half of s to the right child. If the sequence a

process starts with has length one, the process computes f on that element. However it must wait for a *go* message before printing the result, and then send a *go* message after the result has been printed. However due to dynamic process creation each process cannot know which process it will receive *go* from and which it will send *go* to. This is handled in essence by passing communication channels from parent to child.

Although the Olympic Torch is most useful when f is complex, for simplicity of presentation we will examine the special case in which f is the identity function. A simple sequential solution would print the elements of s in order. Let us now look at the parallel solution.

5.2 Parallel Olympic Torch

The parallel implementation of Olympic Torch involves number of processes, each of which either prints a single element of the sequence or creates two other processes, passing each one half of the problem. See Figure 5-1. There are infinitely many processes: For each $l \geq 0$ process $\mathcal{S}_{l,n}$ exists for $0 \leq n < 2^l$. For each process l indicates the level of the process and n its number in that level; n will also be used as a sort of mail address for the process. In practice we will need only up to level $\lceil \lg(|s|) \rceil$, where s is the sequence to be printed.

All processes are initially "dead"; that is, they must be created via the *start* input action. For all processes $\mathcal{S}_{l,n}$ set $w_{\mathcal{S}_{l,n}}(a) = a.working$. It is easy to see that $\{\mathcal{S}_{l,n}\}$ is a dynamic system (see Section 3).

The process creation action is *start*, which creates a set of processes (in this case at most two). For each process to be created there is a tuple (l, n, r, s, m), where l and n denote the level and number of the process to be created, and r, s, m are initialization parameters. The argument of a *start* output action is a set of two tuples, one for each process to be created. Each process $\mathcal{S}_{l,n}$ has every start action that can create $\mathcal{S}_{l,n}$ (all start(S) where $(l, n, r, s, m) \in S$ for some r, s, m) as an input action.

Precisely describing which *start* actions are in in$(\mathcal{S}_{l,n})$ is slightly complicated. The actions are listed as $start(\{(l, n, r, s, m)\} \cup b_{l,n})$. By this it is meant that $start(b)$ is an input action of $\mathcal{S}_{l,n}$ iff $b = \{(l, n, r, s, m)\} \cup b_{l,n}$, where r is a boolean value, s is a finite nonempty sequence, and m is an integer and $b_{l,n} \in B_{l,n}$. If $(l, n) \neq (0, 0)$ then $B_{l,n} = \bigcup_{r',s',m'} \{\{(l, n', r', s', m')\}\}$, where r', s', m' are appropriately typed, and $n' = n + 2^{l-1}$ if $n < 2^{l-1}$; $n' = n - 2^{l-1}$ otherwise.

State Variables:

working: *boolean* (initially *false*)

ready: *boolean* (initially *false*)

seq: *sequence* (initially ϵ)

next: *integer* (initially 0)

Input Actions:

$start(\{(l,n,r,s,m)\} \cup b_{l,n})$ [see text for conditions]

effects:

if $working = false$ then

$working \leftarrow true$

$ready \leftarrow r$

$seq \leftarrow s$

$next \leftarrow m$

$go_{l',n'}(n)$ [for all $l' \geq 1$ and $0 \leq n' \leq 2^l - 1$ except $(l',n') = (l,n)$]

effects:

if $working = true$ then $ready \leftarrow true$

Output Actions: (all in one partition)

$start(\left\{(l+1, n, ready, \text{lefthalf}(seq), n+2^l), (l+1, n+2^l, false, \text{righthalf}(seq), m)\right\})$

preconditions:

$working = true$

$|seq| > 1$

effects:

$working \leftarrow false$

$go_{l,n}(next)$

preconditions:

$working = true$

$ready = true$

$seq = \epsilon$

effects:

$working \leftarrow false$

$print_{l,n}(s_1)$

preconditions:

$working = true$

$ready = true$

$seq = [s_1]$

effects:

$seq \leftarrow \epsilon$

Figure 5-1: States and actions of $S_{l,n}$, $l \geq 0, 0 \leq n < 2^l$.

State Variables:

queue: *sequence* (initially ϵ)

Input Actions:

$print_{l,n}(a)$ for all $l \geq 0$, $0 \leq n < 2^l$

effects:

$queue \leftarrow queue \cdot a$

Output Actions:

$print(a)$

preconditions:

$|queue| \geq 1$

$a = \text{head}(queue)$

effects:

$queue \leftarrow \text{tail}(queue)$

Figure 5-2: Printer Front End $\mathcal{F}$

lefthalf$(s) = s_1 \ldots s_k$

righthalf$(s) = s_{k+1} \ldots s_m$

where $m = |s|$ and $k = \lfloor m/2 \rfloor$

Figure 5-3: Supplementary Functions

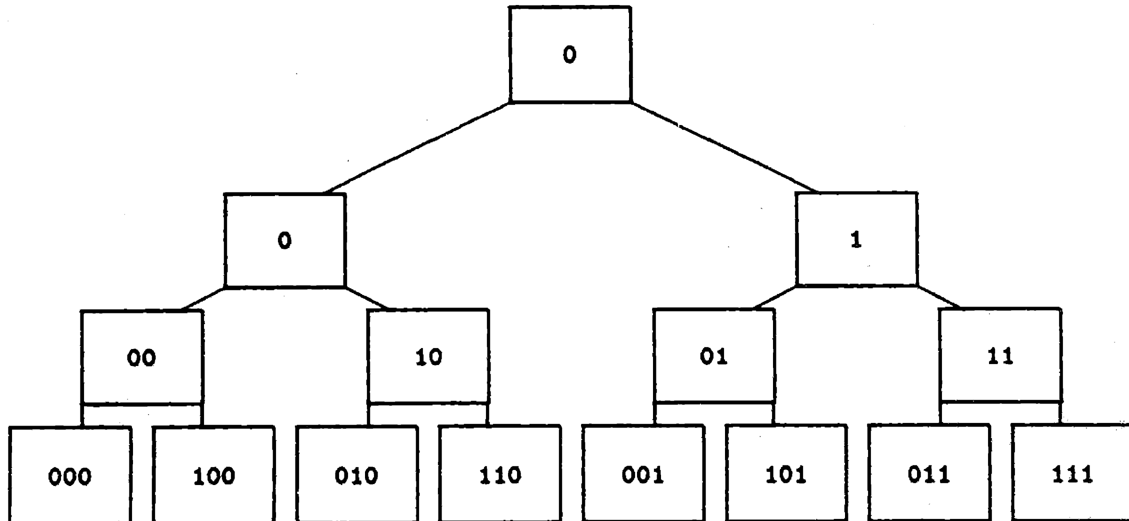

Figure 5-4: Process Creation Tree

In other words $b_{l,n}$ represents the tuple corresponding to the other process created by that start action. If $(l, n) = (0,0)$, then let $B_{l,n} = \{\emptyset\}$, so no other process will be created when the root process $\mathcal{S}_{0,0}$ is created. Therefore the output actions of the environment of Olympic Torch will be $start(\{(0, 0, r, s, m)\})$ for appropriate r, s, m.

Since the effects of input action $start(\{(l, n, r, s, m)\} \cup b_{l,n})$ do not depend upon $b_{l,n}$, for conciseness this action is sometimes referred to simply as $start(l, n, r, s, m)$. Furthermore $start(l, n)$ is used to refer to $start(l, n, r, s, m)$ for arbitrary r, s, m.

It's easy to see that process $\mathcal{S}_{l,n}$ creates exactly the processes $\mathcal{S}_{l+1,n}$ and $\mathcal{S}_{l+1,n+2^l}$. The former will be referred to as the *left child* and the latter the *right child.* Notice that a left child always has $n < 2^{l-1}$ and a right child always has $n \geq 2^{l-1}$. Also each process (except $\mathcal{S}_{0,0}$) has a unique parent and the parent of $\mathcal{S}_{l,n}$ is $\mathcal{S}_{l-1,n \bmod 2^{l-1}}$. See Figure 5-4.

Synchronization is handled using the variables *next* and *ready*. When a process creates two children, the "mail address" n of the left child will be the same as the parent. The *next* of the right child will be the same as the parent, and the *next* of the left child is set to the mail address of the right child. This can be viewed as passing imaginary communication channels between processes. Thus the parent passes its incoming channel to the left child, its outgoing channel to the right child, and creates a new channel between the two. The variable *ready*, when true, indicates that the process is able to print. When a *go* message is sent to the process *ready* will be set to be true; the process will then pass this value to its left child if it creates processes rather than prints.

Note that output actions are parameterized by state variables (which may change value), so it is somewhat ambiguous exactly what out($\mathcal{S}_{l,n}$) is. For simplicity we assume that an action exists for every possible parameter of that type. The state variables then serve to indiate which output actions are enabled from that state.

Notice that there are two supplementary functions used in the definition of $\mathcal{S}_{l,n}$. See Figure 5-3. To divide a sequence into two parts we use lefthalf and righthalf; these could actually be used for any $k \in \{1, \ldots, |s| - 1\}$ but not dividing nearly in half means the process creation tree will have a greater depth, as much as $|s|$.

Due to the restriction that output actions of I/O Automata be disjoint, we cannot simply have $print(a)$ actions for each process. To get around this we subscript each $print(a)$ action

as $print_{l,n}(a)$. One could then add a front end to the printer that converts a $print_{l,n}(a)$ input action into a $print(a)$ output action such as that shown in Figure 5-2. However we won't be concerned with subscripts here, and will just assume the printer throws them away. Therefore $print(a)$ will be used to refer $print_{l,n}(a)$ for arbitrary l and n.

In addition, since more than one process could have $go(next)$ as an output action, we will subscript the go input and output actions with l and n as well. Note that since the go input action is not allowed to have (l, n) as a subscript, each automaton $\mathcal{S}_{l,n}$ is well-defined. We will use $go(next)$ to refer to $go_{l,n}(next)$ for arbitrary l and n.

Let C be the composition of the $\mathcal{S}_{l,n}$ for $l \geq 0$ and $0 \leq n < 2^l$. It is easy to see, due to the subscripting of the $print$ and go actions, that all the automata composing C are compatible. In fact they are strongly compatible, since no automaton has internal actions. Let Σ be the set of all $start$ output actions of C along with all $go(m)$ actions where $m \geq 0$. Then let $C' = \text{Hide}_\Sigma(C)$. Note that the only input actions of C (and C') are $start(\{(0, 0, r, s, m)\})$ for some r, s, m.

5.3 Correctness

Before proving correctness, it is useful to characterize the environment of the Olympic Torch. Save for certain rely conditions, we are not concerned with its behavior, so we will only specify an action signature for the environment E. The input actions of E will be all $print$ actions and all $go(m)$ actions with with $m < 0$. The output actions of E will be all actions $start(\{(0, 0, true, s, m)\})$ where s is a finite nonempty sequence and $m < 0$.

We can state the required correctness conditions for Olympic Torch as an external schedule module with a rely/guarantee function. The external action signature of this module P will be the same as C'. Let $s = s_1 s_2 \ldots s_q$ be the sequence to be printed. Let m be some negative integer. Then

$$
\begin{aligned}
r_{s,m} &= [start(0, 0, true, s, m)] \\
G_P(r_{s,m}) &= \{r_{s,m} \cdot print(s) \cdot [go(m)]\}
\end{aligned}
$$

where $print(s) = [print(s_1), print(s_2), \ldots, print(s_q)]$. For all other $r \in \text{S}(\text{in}(P))$, define $G_P(r) = \text{S}(\text{acts}(P))$. In other words, the behavior could be anything and we need not worry about it.

We are only concerned with the set $R = \bigcup_{s,m} \{r_{s,m}\}$. Therefore to show that C' is correct, we need to show $G_{C'}(r) = G_P(r)$ for all $r \in R$.

To show correctness, note that each automaton $\mathcal{S}_{l,n}$ is essentially the same. Let $\mathcal{C}_{l,n}$ (for $l \geq 0$ and $0 \leq n < 2^l$) be the composition of the set of processes in $\mathcal{C}$ consisting of $\mathcal{S}_{l,n}$ and all its descendants (thus $\mathcal{C}_{0,0} = \mathcal{C}$). We will give a rely/guarantee function for each tree, but we first want to hide actions that are internal to the tree (see Section 2.4.1).

Let $\overline{\mathcal{C}}_{l,n}$ be the composition of all processes not in $\mathcal{C}_{l,n}$, so that $\mathcal{C}_{l,n} \cdot \overline{\mathcal{C}}_{l,n} = \mathcal{C}$. Define $\Psi_{l,n} = \text{acts}(\mathcal{C}_{l,n}) \cap \text{acts}(\overline{\mathcal{C}}_{l,n})$. The input actions of $\mathcal{C}_{l,n}$ that are in $\Psi_{l,n}$ are the $start(l, n \bmod 2^{l-1})$ actions and some $go(m)$ actions where $m > 0$ and $m \bmod 2^l = n$; the output actions are some $go(m)$ actions for $m > 0$. Let $\Sigma_{l,n} = \Sigma - \Psi_{l,n}$, where Σ was the set of actions hidden to produce $\mathcal{C}'$. Let $\mathcal{C}'_{l,n} = \text{Hide}_{\Sigma_{l,n}}(\mathcal{C}_{l,n})$ and define $\overline{\mathcal{C}}'_{l,n}$ similarly. It then follows $\text{Hide}_{\Psi_{l,n}}(\mathcal{C}'_{l,n} \cdot \overline{\mathcal{C}}'_{l,n}) = \mathcal{C}'$ (from Theorem 2.14) and also $\mathcal{C}'_{0,0} = \mathcal{C}'$.

It is important for what follows to compose a parent with the process creation trees of its two children. The following lemma shows how this can be done.

Lemma 5.1 *Let A be the set of all* start *output actions of $\mathcal{S}_{l,n}$. Let B be the set of all* go(m) *actions that are an output of one of the three processes $\mathcal{S}_{l,n}, \mathcal{C}'_{l+1,n}, \mathcal{C}'_{l+1,n+2^l}$ and an input of one of the other two. Let $\Gamma = A \cup B$. Then $\mathcal{C}'_{l,n} = \text{Hide}_\Gamma(\mathcal{S}_{l,n} \cdot \mathcal{C}'_{l+1,n} \cdot \mathcal{C}'_{l+1,n+2^l})$.*

Proof: The set Γ describes exactly those actions which are in the action signatures of two of the three automata in question. Furthermore $\text{Hide}_{\Sigma_{l,n}-\Gamma_{l,n}}(\mathcal{S}_{l,n}) = \mathcal{S}_{l,n}$ and similarly for $\mathcal{C}'_{l+1,n}$ and $\mathcal{C}'_{l+1,n+2^l}$. The lemma then follows from Theorem 2.14. □

We will specify the rely/guarantee function for $\mathcal{C}'_{l,n}$ as the following function for a schedule module $P_{l,n}$ whose external action signature is the same as that of $\mathcal{C}'_{l,n}$. Its rely/guarantee function is as follows, where $s = s_1 s_1 \ldots s_q$ and either $m < 0$ or $m \not\equiv n \bmod 2^l$.

$$\begin{aligned} r^1_{s,m}(l,n) &= [start(l,n,true,s,m)] \\ r^2_{s,m}(l,n) &= [go(n), start(l,n,true,s,m)] \\ r^3_{s,m}(l,n) &= [start(l,n,false,s,m), go(n)] \\ G_{P_{l,n}}(r^i_{s,m}(l,n)) &= \left\{ r^i_{s,m}(l,n) \cdot print(s) \cdot [go(m)] \right\} \end{aligned}$$

Rely condition $r^1_{s,m}$ is simply the analog of the previous $r_{s,m}$. The other two rely conditions involve the *go* action, which should cause the sequence to be printed correctly whether it arrives

before the *start* action or not. Note that $go(m)$ is an output action of $P_{l,n}$ only if $m < 0$ or $m \not\equiv n \bmod 2^l$; thus the restriction.

In addition we to specify some additional safety properties of the guarantees. Define $R_{l,n}$ to be the set of all $r \in \mathrm{S}(\mathrm{in}(P_{l,n}))$ such that there is exactly one $start(l,n)$ action in r. Say that this action is $start(l,n,b,s,m)$ for some b, s, m. Then $z \in G_{P_{l,n}}(r)$ only if no output action of $P_{l,n}$ occurs before this *start* action. Also there may be at most one *go* output action of $\mathcal{P}$ in z, and it must be $go(m)$ if it occurs. This will be used to prove that the process creation trees of the children of $\mathcal{S}_{l,n}$ will not make any transitions until they are created, and that they will not send any superfluous *go* messages.

For all other $r \in \mathrm{S}(\mathrm{in}(P_{l,n}))$, define $G_{P_{l,n}}(r) = \mathrm{S}(\mathrm{acts}(P_{l,n}))$ as before. Note that $P_{0,0}$ solves P; therefore if we can show that C' solves $P_{0,0}$ it follows it also solves P. This will be done by more generally showing that $C'_{l,n}$ solves $P_{l,n}$ for all l and n.

Lemma 5.2 *Automaton $C'_{l,n}$ solves $P_{l,n}$ for all $l \geq 0$ and $0 \leq n < 2^l$.*

Proof: The only interesting cases are for the sequences $r^i_{s,m}(l,n)$ for $1 \leq i \leq 3$, as well as the safety properties for $R_{l,n}$. The proof is by induction on $q = |s|$. This means it is essentially induction on the size of the process creation tree rooted at $\mathcal{S}_{l,n}$.

The base case is $q = 1$. In this case, for each of the r^i, all local actions of $C'_{l,n}$ will be disabled until the actions of r^i have occurred. At this point only $print_{l,n}(s_1)$ will be enabled; it is continuously enabled and by fairness it must occur. Next $go_{l,n}(m)$ will be continuously enabled and thus must occur. After it occurs no action will be enabled. It is clear that the safety properties for $R_{l,n}$ are satisfied as well.

In the induction step $q > 1$; let $s^1 = \mathrm{lefthalf}(s)$ and $s^2 = \mathrm{righthalf}(s)$, so $s^1 \cdot s^2 = s$ and $|s^i| < q$ for $i \in \{1,2\}$. Let $B(y) = \left\{(l+1, n, y, s^1, n+2^l), (l+1, n+2^l, \mathit{false}, s^2, m)\right\}$.

We can handle the safety properties for $R_{l,n}$ separately, and use them later when needed. For $r \in R_{l,n}$ there is exactly one $start(l,n)$ input action in r; say that it is $start(l,n,b,s,m)$. No local action of $\mathcal{S}_{l,n}$ is enabled until this action occurs. At this point only $start(B(y))$ is enabled for some y; once it occurs no local action of $\mathcal{S}_{l,n}$ is enabled and none will since by $R_{l,n}$ there is only one $start_{l,n}$ action. Therefore $R_{l+1,n}$ and $R_{l+1,n+2^l}$ are satisfied for the process creation trees of the two children; by induction no action of either will occur before the $start(B(y))$

action, and the only *go* output actions will be $go(n+2^l)$ and $go(m)$. The former is hidden in $C'_{l,n}$, so the only *go* output action of $C'_{l,n}$ will be $go(m)$. Thus the safety properties are satisfied.

We now consider the case of $r^i_{s,m}(l,n)$, and first describe $G_{S_{l,n}}(r^i)$. In the case of each r^i, no local action of $C'_{l,n}$ will be enabled until the *start* input action occurs; following this (in cases r^1 and r^2) the $start(B(true))$ output action will be the only action enabled; it will be continuously enabled and thus must occur. The case r^3 is slightly more complicated. Once the *start* input action occurs $start(B(false))$ will be the only action enabled. It may occur before the $go(n)$ action; otherwise after $go(n)$ occurs the only action enabled will be $start(B(true))$, which will be continuously enabled and thus must occur. In summary:

$$\begin{aligned} G_{S_{l,n}}(r^i) &= \left\{r^i \cdot [start(B(true))]\right\} \text{ (for } i \in \{1,2\}) \\ G_{S_{l,n}}(r^3) &= \left\{r^3 \cdot [start(B(true))], [start(l, n \bmod 2^{l-1}, false, s, m), start(B(false)), go(n)]\right\} \end{aligned}$$

Furthermore it can be seen that if r^i is a subsequence of the inputs and there are no other $start(l,n)$ actions, then the sole output action of $S_{l,n}$ will be the corresponding $start(B(y))$ action and it will follow $start(l,n)$; this is a safety property for $S_{l,n}$.

We can apply the induction hypothesis to $C'_{l+1,n}$ and $C'_{l+1,n+2^l}$ to determine the rely/guarantee functions for them. Let $C = S_{l,n} \cdot C'_{l+1,n} \cdot C'_{l+1,n+2^l}$. Then we can determine the rely/guarantee function for C using Theorem 2.12. Let $Q = print(s^1) \cdot [go(n+2^l)] \cdot print(s^2) \cdot [go(m)]$. We will show that

$$\begin{aligned} G_C(r^i) &= \left\{r^i \cdot [start(B(true))] \cdot Q\right\} \text{ (for } i \in \{1,2\}) \\ G_C(r^3) &= \left\{r^3 \cdot [start(B(true))] \cdot Q, [start(l, n, false, s, m), start(B(false)), go(n)] \cdot Q\right\} \end{aligned}$$

Since the $start(B(y))$ and $go(n+2^l)$ actions are hidden in $C'_{l,n}$ it will then follow from Lemma 5.1 and Theorem 2.15 that $C'_{l,n}$ solves $P_{l,n}$.

We will prove the rely/guarantee function for C is as given above using Theorem 2.12. Given $z \in \mathbf{S}(\mathrm{ext}(C))$ such that $z|\mathrm{in}(C) = r^i$, we show that $z \in G_C(r^i)$ (given above) iff $z|\mathrm{in}(A) \in G_A(Z|\mathrm{in}(A))$ for all $A \in \left\{S_{l,n}, C'_{l+1,n}, C'_{l+1,n+2^l}\right\}$. The "only if" direction is trivial to show; we now prove that the "if" direction holds.

The safety properties simplify matters considerably. Since there is exactly one $start(l,n)$ action in z, from the safety property for $S_{l,n}$ it follows that the sole output action of $S_{l,n}$ will be

the $start(B(y))$ action. From the safety properties for $C'_{l+1,n}$ and $C'_{l+1,n+2^l}$ it follows any local action of either must follow this action, and that the only *go* output actions of either may be $go(n+2^l)$ for the former and $go(m)$ for the latter. Note that $go(n+2^l)$ is an input action of $C'_{l+1,n+2^l}$, and that $go(m)$ by hypothesis is not an input action of any of the three automata.

Therefore $z|\text{in}(S_{l,n}) = r^i_{s,m}(l,n)$, and $G_{S_{l,n}}(r^i_{s,m})$ is as described above. From this it follows $z|\text{in}(C'_{l+1,n}) = r^j_{s^1,n+2^l}(l+1,n)$, where $j=i$ if $i \in \{1,2\}$; $j \in \{2,3\}$ otherwise. The guarantees for this are given by the induction hypothesis. From this it follows $z|\text{in}(C'_{l+1,n+2^l}) = r^k_{s^2,m}(l+1,n+2^l)$, where $k \in \{2,3\}$; the guarantees are again given by the induction hypothesis. Taking the conjunction of each possible triplet of guarantees, it is easily seen that $z \in G_C(r^i_{s,m}(l,n))$. This concludes the proof. □

Theorem 5.3 *Automaton C' solves the Olympic Torch problem.*

Proof: Since $C' = C'_{0,0}$ and $P_{0,0}$ solves P, this follows directly from Lemma 5.2. □

5.4 Comments

A previous version of this proof used a style similar to specification using a rely/guarantee function, but was somewhat less formally done with a separate proof of safety conditions. It can be seen here, just as in the Set Partition Example, that the rely/guarantee conditions capture both safety and liveness conditions, thus allowing a unified proof. The result is that this proof is considerably shorter than the previous version.

De Champeaux ([dC89]) gets into real trouble in his proof of Olympic Torch, due to his lack of an underlying operational model. He does not send results to the printer, but instead simply wants to show that the function f is evaluated on each element of the sequence (actually a binary tree in his paper) in order. But there is no way for him to define this, so he has to add a global variable that is accessed by each process when it evaluates f (thus acting as a piece of paper being printed on). The complications arising from adding this global variable are not addressed. Also the same problem with the Set Partition proof is present here: it is only shown that the correct behaviors are a subset of the behaviors of the algorithm. It is worth noting that his proof of Olympic Torch is much shorter and simpler than his proof of Set Partition, as was

true to some extent here. This is an indication that, in determining how difficult an algorithm is to prove correct, its structure is more important than whether it uses process creation.

Chapter 6

Actors

The examples Set Partition and Olympic Torch give a good feeling for how process creation and dynamic topologies can be modeled in a straightforward way using I/O automata. However they are only specific algorithms. We now look at an entire programming methodology, that of Carl Hewitt's Actor model ([Agh86, Cli81]). Programming in Actors depends heavily on process creation and a dynamic topology, which led Agha to claim that Actors are somehow more powerful than a static model such as CSP ([Agh86], p. 9). We show here that this claim is false by defining actors in terms of I/O Automata, an apparently static model. Of course a programming language based on the Actor model may be more convenient for writing programs that take advantage of creation and dynamic topologies, but by defining Actors in terms of I/O Automata we gain a rigorous framework in which correctness proofs can be done.

First an example (recursive factorial) will be used to give an informal description of Actors. Then Agha's operational semantics will be given. Following this, two I/O automata definitions of Actors will be given, one using asynchronous message passing, and the other using instantaneous message passing. The fair behaviors of these automata will the shown to be the same, and the latter automaton will be shown to be equivalent to Agha's definition.

With this established, we prove correctness for recursive factorial in the Actor model. The proof style is similar to one that could be used in I/O automata. For more complicated examples it might be advantageous work in the I/O automaton model, since this is a well-established model for which many proof techniques have been developed. However there are currently difficulties in doing this which are discussed at the end of the chapter.

```
def Rec-Factorial () [n, c]
become Rec-Factorial
if n = 0 then
  then send [1] to c
  else let c' = new Rec-Customer(n, c)
           {send [n - 1, c'] to self}
```

```
def Rec-Customer (n, c) [k]
send [n * k] to c
```

Figure 6-1: Recursive Factorial Behaviors

6.1 Example: Recursive Factorial

An actor can be thought of informally as a process that receives messages, and in response sends other messages, creates other actors, and changes its behavior. Before diving into a formal model of actors, we will look at an example to get a more concrete idea of what actors are intended to be. A popular example of an Actor program is the recursive factorial. An informal implementation of recursive factorial is given on pages 35–36 of [Agh86]; an implementation in SAL (a Simple Actor Language presented in Agha's book) is given in Figure 6-1. Actually Agha's definition of SAL is extremely vague and it is not always clear what the syntax is intended to mean; however for this example at least we will attempt to be more precise.

An *actor* can be regarded as an entity whose state is completely characterized by a *mail address* (also known as a *handle*) and a *behavior*. A behavior is a function that takes as input a *task* (informally the incoming message) and produces as output a triple consisting of a finite set of tasks (messages sent to other actors), a finite set of actors (newly created actors), and a replacement behavior. A task is a triple consisting of a *tag* which uniquely identifies it, a *target* which is the mail address the task is to be delivered to, and a *communication* which is the message to be delivered. A *mail system* is assumed to exist which delivers tasks to their targets. This mail system must deliver all and only those tasks sent, and the tasks are delivered in an arbitrary order and after an arbitrary delay. Note that since the behavior of an actor is just a function, actors are "event-driven"; that is, when a task is delivered it is immediately processed, so "delivery" and "processing" are really synonymous.

In SAL, behaviors are defined using **def** statements, and two such behaviors are given in Figure 6-1, for *Rec-Factorial* and *Rec-Customer*. The name of a behavior is followed by

an *acquaintance list* (in parenthesis), which is essentially a set of state constants (defined at creation time), and may include mail addresses of other actors that an actor with this behavior can communicate with. For example the acquaintance list of *Rec-Customer* consists of an integer n and a mail address c. (Note that typing is not part of the SAL language, although Agha suggests it could be easily added.) The acquaintance list of *Rec-Factorial* is empty.

Following the acquaintance list is a *communication list* describing the communications (inputs) the behavior can handle. These communications are of course the third component of some received task, and are represented as a tuple enclosed in square brackets. The acceptable communication for *Rec-Factorial* is a pair consisting of an integer n and a mail address c; the communication for *Rec-Customer* is an integer k.

Following the communication list is a sequence of commands, which are intended to be executed concurrently. There are three major kinds of commands corresponding to the output triple of a behavior. A **send** command is used to create new tasks. A **new** command is used to create new actors. Finally a **become** command is used to specify the replacement behavior. So an actor with the *Rec-Factorial* behavior, upon processing a task with communication $[n, c]$, becomes an actor with the same behavior. At the same time, it either creates a task with c as the target and [1] as the communication (in the case $n = 0$); or else it creates a new actor c' whose behavior is *Rec-Customer* with acquaintances n and c, and also creates a new task with *self* (the mail address of the actor itself) as the target and $[n-1, c]$ as the communication. An actor with the *Rec-Customer* behavior and acquaintances n and c, upon processing a task with communication k, simply creates a new task with c as the target and $n*k$ as the communication. It creates no new actors, and note that a replacement behavior is not specified. Agha states (p. 33) that the replacement behavior, if there is no **become** statement, is the same as the current behavior. Therefore the replacement behavior for any processed task for both *Rec-Factorial* and *Rec-Customer* is the same as the current behavior.

To see how such an actor program would execute, let's compute the factorial of three. We must first specify an initial *configuration*, which is a pair consisting of a finite set of actors and a finite set of tasks. To specify a top-level actor in SAL, one uses a command such as

$$\texttt{let } r = \texttt{new } \textit{Rec-Factorial}()$$

Here r is the mail address of our recursive factorial actor, which will be referred to as R.

We'll also assume the existence of some customer (to receive the result of the computation) whose mail address is u. There seems to be no way in SAL to specify an initial set of tasks (therefore making it impossible to start any computation!) but we will assume an initial task $\tau = (r, [3, u])$. For simplicity we will omit the tag of a task here, so r is the target of τ and $[3, u]$ is the communication.

When R processes task τ, it checks to see if $3 = 0$; since this is false, it creates a new actor C_1 with mail address c_1, behavior *Rec-Customer*, and acquaintances 3 and u. It also sends itself the task $(r, [2, c_1])$. When R receives this task, it creates a new actor C_2 with mail address c_2, behavior *Rec-Customer*, and acquaintances 2 and c_1. It also sends itself the task $(r, [1, c_2])$. When R receives this task, it creates a new actor C_3 with mail address c_3, behavior *Rec-Customer*, and acquaintances 1 and c_2. It also sends itself the task $(r, [0, c_3])$. When it receives this task, it sends the task $(c_3, [1])$. Now when C_3 processes the task sent to it, it sends task $(c_2, [1])$. Next C_2 upon processing this task will send task $(c_1, [2])$. Finally C_1 upon processing this task will send task $(u, [6])$, completing the computation of 3!. Note that in general the mail system may deliver tasks out of order, but as each actor here receives and sends at most one message the computation ends up being sequential.

6.2 Agha's Operational Semantics for Actors

In this section we present the operational semantics for Actors given in Chapter 5 of [Agh86]. In the next section we will show how this semantics can be specified using I/O Automata. Many of the terms in this section were defined informally in the previous section. We now present them again formally. Often the definitions Agha gives are vague or contradictory; the definitions here have been corrected accordingly.

6.2.1 Basic Definitions

Surprisingly, Agha gives no formal definition of an *actor*; and what's more surprising, such a definition is not needed to formally define an overall actor system. This is because the elements that make up an actor: its mail address, its behavior, and its transitions, are all present in the definition of an actor system given below; however they are not presented as a conceptual whole.

The mail address and behavior will be collected together as an object called an *actor state*, and the transitions of an actor are a part of the transitions of the actor system. Although actor won't be given a formal definition, it will often be used informally in the following, and can be thought of as an automaton whose state is an actor state, and whose transitions correspond to tasks that have the mail address of the actor as a target (these terms are defined below).

An *actor system* can be thought of as a state machine with an infinite number of states and an infinite number of transitions. The transitions will be described later. A state of an actor system is called a *configuration*, and it consists of a finite set of *actor states* and a finite set of *tasks*, which are the messages sent but not yet processed by their targets. To distinguish otherwise identical tasks, each one has a unique *tag*. Similarly every actor has a unique *mail address*. Agha specifies a form for both of these which will also be used here: a tag or mail address is a finite sequence of natural numbers. Agha writes such a sequence as $i_1.i_2.i_3$ rather than $[i_1, i_2, i_3]$; his notation will be followed in this chapter.

The set of all possible tasks, $\mathcal{T}$, is then defined to be $\mathcal{I} \times \mathcal{M} \times \mathcal{K}$, where $\mathcal{I}$ is the set of all possible tags, $\mathcal{M}$ is the set of all possible mail addresses, and $\mathcal{K}$ is the set of all possible communications, where a communication is an arbitrary tuple of values. If $\tau = (t, m, k)$ is a task then t is the *tag* of τ, m is the *target*, and k is the *communication*. The finite set of tasks in configuration c is denoted tasks(c).

Let $\mathcal{A}$ be the set of actor states (incorrectly called the set of actors by Agha). Each actor state $A \in \mathcal{A}$ has associated with it a mail address mail(A) $\in \mathcal{M}$ and a behavior beh(A) $\in \mathcal{B}$, where $\mathcal{B}$ is to be defined. We will generally represent an actor state A as a pair (m, β), where m = mail(A) and β = beh(A). The finite set of actor states in a configuration is denoted actors(c).[1]

For a set S, let $F_s(S)$ be the set of all finite subsets of S. Then set of all possible configurations of an actor system is defined to be the set of all pairs (C, T), where $C \in F_s(\mathcal{A})$ and $T \in F_s(\mathcal{T})$, satisfying the following two restrictions:

1. No task in T has a tag which is the prefix of either another tag of a task in T or of a mail address of an actor state in C.

[1] Agha's term for actors(c) is states(c); however we will be using states to refer to the set of configurations.

2. No mail address of an actor state in C is the prefix of either the mail address of another actor state in C or of a tag of a task in T.

These conditions are really stronger than are needed (the minimal requirement is that each tag or mail address is unique, which is implied by the two conditions) but are used by Agha to show that the next configuration (after a transition) will also be well-defined with respect to these conditions.

6.2.2 Actor Behaviors

Agha gives two contradictory definitions of $\mathcal{B}$, the set of behaviors, on the same page ([Agh86], p. 74) so the definitions here will be fixed a bit. He first describes $\mathcal{B}$ as "the set of all possible behaviors" and then defines it as the set of all behaviors for a certain mail address m. There are two ways to fix this: One is to let the latter be $\mathcal{B}_m$, and define $\mathcal{B} = \bigcup_{m \in \mathcal{M}} \mathcal{B}_m$; the other is to define a behavior so that it takes a mail address as an argument (rather than fixing the mail address as a constant). The second approach will be adopted here as it seems closer to Agha's intentions. One might worry about inconsistency if an actor were to process a task that has a different mail address (target) than the actor, but the mail system will ensure this never occurs.

We first define a superset of the set of behaviors.

$$\overline{\mathcal{B}} = (\mathcal{T} \rightarrow F_s(\mathcal{T}) \times F_s(\mathcal{A}) \times \mathcal{A})$$

Informally, if A is an actor whose behavior is $\beta \in \overline{\mathcal{B}}$, then A, upon processing an input task using β, creates (sends) a finite number of tasks, creates a finite number of actors, and replaces the behavior of A with some new behavior β'. The reason $\overline{\mathcal{B}}$ is not itself the set of behaviors, is that in a real actor system there must be additional restrictions to ensure configurations are well-defined, in particular that the tags of newly-created task and mail addresses of newly-created actors are distinct from each other and all previous tags and mail addresses. Also the replacement actor state must have the same mail address as the current actor state. In addition, we will at this point impose the restriction that the set of behaviors be countable. This is satisfied by any real actor system, since such a system will be determined by some

language such as SAL. Furthermore we need such a restriction for the I/O automata semantics in the next section.

Therefore the set of behaviors $\mathcal{B}$ is a countable subset of $\overline{\mathcal{B}}$, such that for any $\beta \in \mathcal{B}$ and all $\tau \in T$, where $\beta(\tau) = (T, C, \gamma)$, the following conditions hold:

1. The tag of τ is a prefix of the tag of every task in T and a prefix of the mail address of every actor state in C.

2. Let I' be the set of tags of the tasks in T and M' be the set of mail addresses of the actor states in C. Then no element of $I' \cup M'$ is the prefix of any other element of the same set.

3. mail(γ) equals the target of τ.

Note that because the set of behaviors is countable, the set of actor states $\mathcal{A}$ will also be countable. We can now define an actor system $\mathbf{A}^{\mathcal{B}}$ with respect to the set of behaviors $\mathcal{B}$. For the remainder of this chapter assume $\mathcal{B}$ is a fixed chosen set, so we will write $\mathbf{A}$ to represent the actor system with $\mathcal{B}$ being implicit. $\mathbf{A}$ is defined by its states and transitions; this will be done explicitly in the next section.

6.2.3 Transitions

Let c_1 and c_2 be two configurations. Then c_1 is said to have a *possible transition* to c_2 by processing a task $\tau = (t, m, k)$, written $c_1 \xrightarrow{\tau} c_2$, if $\tau \in$ tasks(c_1), and furthermore, if actors(c_1)(m) $= \beta$, where

$$\beta(t, m, k) = (T, A, \gamma)$$

then the following hold:

$$\begin{aligned} \text{tasks}(c_2) &= (\text{tasks}(c_1) - \{\tau\}) \cup T \\ \text{actors}(c_2) &= (\text{actors}(c_1) - \{(m, \beta)\}) \cup A \cup \{\gamma\} \end{aligned}$$

He then proves that c_2 is in fact a configuration, in particular that the requirements for tags and mail addresses are met.

We can in fact use terminology closer to that used in I/O automata, which should make the similarity between an actor system and the I/O automaton representation to be defined in the

next section more obvious. With respect to our actor system $\mathbf{A}$, let states($\mathbf{A}$) be the set of all possible configurations of $\mathbf{A}$ as defined previously. Let

$$\text{steps}(\mathbf{A}) = \left\{(c, \tau, c') : c, c' \in \text{states}(\mathbf{A}), \tau \in \mathcal{T}, \text{ and } c \xrightarrow{\tau} c'\right\}$$

A task τ is said to be *enabled* from a configuration c if there exists some c' for which $(c, \tau, c') \in$ steps($\mathbf{A}$).

Define an *execution* of $\mathbf{A}$ to be a (finite or infinite) alternating sequence of configurations and tasks $c_0\tau_1c_1\tau_2c_2\ldots$ such that $(c_i, \tau_{i+1}, c_{i+1}) \in$ steps($\mathbf{A}$) for all $i > 0$. Note that due to the fact that tags are required to be distinct, $c_i \neq c_j$ and $\tau_i \neq \tau_j$ for all i and j. Configuration c_0 is called the *initial configuration* of the execution fragment. There is no restriction as to which configurations be initial configurations of $\mathbf{A}$. Therefore we can define the set of start states start($\mathbf{A}$) = states($\mathbf{A}$). Denote the set of all executions of $\mathbf{A}$ as execs($\mathbf{A}$) . For an execution x, define sched(x) to be the subsequence of tasks in x. These definitions correspond to those for I/O automata.

Agha also specifies the needed condition to guarantee mail delivery, the only liveness condition he requires. By "mail delivery" he means that every task created in an actor system is eventually delivered. Given configurations c and c' and a task $\tau \in$ tasks(c), c is said to have a *subsequent transition* to c' with respect to τ, written $c \overset{\tau}{\hookrightarrow} c'$, if there exist finite $x, y \in$ execs($\mathbf{A}$) such that c is the initial configuration of x and $y = x \cdot \tau c'$. (Agha phrases this in a different but equivalent manner.) Thus c' represents the first configuration after τ is processed. Now let $X_c^\tau = \left\{c' : c \overset{\tau}{\hookrightarrow} c'\right\}$. Then Agha says "the guarantee of mail delivery implies that the configuration c must pass through a configuration in X_c^τ" (p. 87). This is vaguely worded, but we can make it precise by defining the set of *fair executions* fair($\mathbf{A}$) of an actor system $\mathbf{A}$. Define fair($\mathbf{A}$) to be the set of all $x \in$ execs($\mathbf{A}$), $x = c_0\tau_1c_1\ldots$, such that for all $i > 0$ and all $\tau \in$ tasks(c_i), τ occurs in x. (Note that τ must occur after c_i due to uniqueness of tasks.) In other words, $x \in$ fair($\mathbf{A}$) iff every unprocessed task in any configuration of x is eventually processed. This is exactly what was meant by "guarantee of mail delvery." Define the fair behaviors fbeh($\mathbf{A}$) to be $\{z : z = \text{sched}(x) \text{ for some } x \in \text{fair}(\mathbf{A})\}$.

6.3 I/O Automata Actors

We now show how an actor system can be defined using I/O automata. One possibility would be to define a single automaton to represent **A**. However for modularity it is perhaps cleaner to define an I/O automaton to correspond to each actor. It then would be desirable to define an additional automaton to correspond to the mail system. However this is not quite possible. Recall that actors are event-driven, meaning that when a task is delivered to an actor, it immediately creates new actors and tasks. However when a message is received by an I/O automaton (which must correspond to an input action), the automaton need not process the message (which corresponds to an output action). Therefore when we model an actor in I/O automata, we must say that the delivery (which is the same as the processing of the task) corresponds to an output action. In this case the automaton must receive and buffer the message separately. In other words, the automaton will also be used to represent part of the mail system, as well as a single actor.

We specify the mail system in two different manners, and define an I/O automata actor system based upon each one. We then show that the fair behaviors of these two systems are the same. Finally we show that the second, simpler, system is equivalent to Agha's system. It in this system in which correctness proofs will be done.

6.3.1 I/O Automata Actors with Separated Mail System

In the first mail system, there will be a separate automaton for the mail system; however we need to include queues at each automaton representing an actor which will also be a part of the mail system. Note that "delivery" of a task will correspond to the event that a task is removed from such a queue, not the event that it is added to the queue. Each I/O automaton corresponding to an actor and its mail queue will be simply referred to as an "actor" automaton, even though this is not technically true.

Define $\mathcal{T}_m = \{\tau \in \mathcal{T} : m \text{ is the target of } \tau\}$. For every mail address $m \in \mathcal{M}$, define an I/O automaton actor A_m as shown in Figure 6-2. Note that the definition depends upon the chosen set of behaviors $\mathcal{B}$, and that A_m has a countable number of states and actions. Each actor A_m has two state variables. The symbol $\perp$ is used to indicate that there is no defined

State Variables:

β_m: an element of $\mathcal{B} \cup \{\perp\}$

$queue_m$: sequence of tasks, each in $\mathcal{T}_m$ (initially ϵ)

Input Actions:

$process_t(T, C)$ [for all $t \in \mathcal{T} - \mathcal{T}_m$, all $T \in F_s(\mathcal{T})$, and all $C \in F_s(\mathcal{A})$ such that $\exists b \in \mathcal{B}$ s.t. $(m, b) \in C$]

effects:

$\beta_m \leftarrow b$

$receive(\tau)$ [for all $\tau \in \mathcal{T}_m$]

effects:

if $\beta_m \neq \perp$ then $queue_m \leftarrow queue_m \cdot \tau$

Output Actions: (all actions with same t are in one class of the partition)

$process_t(T, C)$ [for all $t \in \mathcal{T}_m$]

preconditions:

$\beta_m \neq \perp$

head$(queue_m) = t$

$\beta_m(t) = T \times C \times (m, b)$

effects:

$queue_m \leftarrow$ tail$(queue_m)$

$\beta_m \leftarrow b$

Figure 6-2: Actor A_m

State Variables:

buf: set of tasks

Input Actions:

$process_t(T, C)$ [for all $t \in \mathcal{T}$, all $T \in F_s(\mathcal{T})$, and all $C \in F_s(\mathcal{A})$]

effects:

$buf \leftarrow buf \cup T$

Output Actions: (all actions are in separate classes of the partition)

$receive(\tau)$

preconditions:

$\tau \in buf$

effects:

$buf \leftarrow buf - \tau$

Figure 6-3: Actor Mail System M

behavior at mail address m. With process creation in mind we can define the "working" function $w_{A_m}(a) = (a.\beta_m \neq \perp)$ for all states a. Since there is always a replacement behavior for any current behavior, it follows that once a actor is alive it will remain alive forever. The variable $queue_m$ is the mail queue for A_m mentioned before.

Each *process* action has two arguments, as well as an index identifying the task it processes. This index serves two purposes. The first is to ensure the automata are well-defined. Because we restrict $queue_m$ to contain only elements of $\mathcal{T}_m$, no two automata will have the same output action, and one automaton will not have the same action as both an input and output action. The second purpose is to ensure fairness with respect to tasks; there is a separate class of the partition of A_m for each index t. This is not strictly needed here because an action when enabled will be continuously enabled (so the partition could be defined as having every action in a separate class), but it will be needed for the simplified actors to follow so it will be used here as well.

The first argument of a *process* action is T, a set of tasks sent to the mail system, which will eventually deliver them to their targets. The second argument is C, a set of actors to be created. They are created instantaneously as specified by the actor model, rather than through the message system as was done with Set Partition and Olympic Torch. When a $process_m$ action occurs the replacement behavior is also instantiated at mail address m.

Note that $\text{start}(A_m) = \{a \in \text{states}(A_m) : a.queue_m = \epsilon\}$. When we specify an initial configuration for an actor system we will specify the initial values of β_m for every m with the restriction that $\beta_m = \perp$ for all but finitely many m. The initial tasks will be in the queue of the mail system.

We now define the mail system M, shown in Figure 6-3. Essentially it extracts tasks from its input actions and stores them in a buffer. Its fair behavior will be to forward every task received after some finite delay. Also note that $\text{start}(M) = \text{states}(M)$. This is because an initial configuration may contain an arbitrary set of tasks.

Let A_M be the composition of all automata in $\{A_m : m \in \mathcal{M}\}$ as well as the mail system M followed by hiding all *receive* actions. Note the correspondence between $\text{states}(A_M)$ and $\text{states}(\mathbf{A})$. If, for $a \in \text{states}(A_M)$, we define $\text{actors}(a) = \{(m, a.\beta_m) : a.\beta_m \neq \perp\}$ and $\text{tasks}(a) = \bigcup_m \{a.queue_m\} \cup a.buf$, then it can be seen there is a surjection from $\text{states}(A_M)$ to $\text{states}(\mathbf{A})$.

State Variables:

β_m: an element of $\mathcal{B} \cup \{\perp\}$

$queue_m$: set of tasks, each in $\mathcal{T}_m$ ($\beta_m = \perp \Rightarrow queue_m = \epsilon$)

Input Actions:

$process_t(T, C)$ [for all $t \in \mathcal{T} - \mathcal{T}_m$, all $T \in F_s(\mathcal{T})$, and all $C \in F_s(\mathcal{A})$ such that $\exists b \in \mathcal{B}$ s.t. $(m, b) \in C$ or $T \cap \mathcal{T}_m \neq \emptyset$]

effects:

if $\exists b \in \mathcal{B}$ s.t. $(m, b) \in C$ then
$\beta_m \leftarrow b$
$queue_m \leftarrow T \cap \mathcal{T}_m$
else if $\beta_m \neq \perp$ then
$queue_m \leftarrow queue_m \cup (T \cap \mathcal{T}_m)$

Output Actions: (all actions are in separate classes of the partition)

$process_t(T, C)$

preconditions:

$\beta_m \neq \perp$
$t \in queue_m$
$\beta_m(t) = T \times C \times (m, b)$

effects:

$queue_m \leftarrow (queue_m - t) \cup (T \cap \mathcal{T}_m)$
$\beta_m \leftarrow b$

Figure 6-4: Simplified Actor A_m

This is not a bijection because an undelivered task with target m could be either in the buffer of M or the queue of A_m. However there is a bijection between start(A_M) and start(**A**). The correspondence between steps(**A**) and steps(A_M) is less obvious, and will be shown instead between steps(**A**) and steps(A), where A is a simplified automaton equivalent to A_M that will be defined in the next section.

6.3.2 Simplified I/O Automata Actors

Automaton A_M seems to be a reasonable operational specification of an actor system. We now present a simpler specification using instantaneous message passing. While this may at first not seem reasonable, we will show that this simpler system A is fairly equivalent to A_M; then that it corresponds exactly to the specification of an actor system given by Agha.

Figure 6-4 shows the simplified automaton A_m for any $m \in \mathcal{M}$. One significant change is that $queue_m$ is now a set rather than a sequence, and it can have any value in a start state

unless $\beta_m = \perp$ (this last condition so that a "dead" actor automaton will have a unique state). Another change is that each *process* input action adds incoming tasks to $queue_m$ as well as handles process creation. The *process* output action also adds tasks to the queue corresponding to those tasks the actor sends to itself. This is done because an automaton cannot have the same action as an input and output action. The partition is unchanged; notice that it is necessary here to ensure fairness to each task, as the automaton may change behaviors which in turn changes the set of actions corresponding to the task. This will be clearer when we show equivalence to Agha's semantics in Theorem 6.2.

Let A be the composition of these simplified automata A_m. For any state $a \in$ states(A), we can define actors(a) $= \{(m, a.\beta_m) : a.\beta_m \neq \perp\}$ (the same as for A_M) and tasks(a) $= \bigcup_m \{a.queue_m\}$. It seems plausible that fbeh(A_M) = fbeh(A), and we will prove this.

Theorem 6.1 fbeh(A_M) = fbeh(A).

Proof:

Define h, a function from states(A_M) to states(A), so that $b = h(a)$ iff actors(b) = actors(a) and tasks(b) = tasks(a). Thus h preserves the configuration implicit in each state. Note that h is well-defined because a task $\tau \in$ tasks(b) is in $queue_m$ iff the target of τ is m. However for any state $b \in$ states(A) for which tasks(b) $\neq \emptyset$, $h^{-1}(b)$ will be a set of states, representing various states of M and orderings of the input queue of each actor. Therefore we will define $h^{-1}(b)$ to be that state a such that $h(a) = b$ and $a.queue_m = \epsilon$ for all $m \in \mathcal{M}$; in other words that state in which no unprocessed tasks have been sent by M.

Let h_1 be the function from states(A_M) to $2^{\text{states}(A)}$ such that $h_1(a) = \{h(a)\}$. Let h_2 be the function from states(A) to $2^{\text{states}(A_M)}$ such that $h_2(a) = \{h^{-1}(a)\}$. Then h_1 is a possibilities mapping from A_M to A, since for any step (a, π, a') of A_M there is a step $(h(a), \pi, h(a'))$ of A if π is a *process* action, and $h(a) = h(a')$ if π is a *receive* action. Also h_2 is a possibilities mapping from A to A_M, since for any step (a, π, a') of A, there is a step $(h^{-1}(a), \gamma, \pi, h^{-1}(a'))$, where γ is the *receive* action for the task that π processes.

Therefore from Theorem 2.7, it follows that for every $x \in$ execs(A_M) there is some $y \in$ execs(A) such that y corresponds to x, and also for every $y \in$ execs(A) there is some $x \in$ execs(A_M) such that x corresponds to y. Furthermore in both cases x is fair iff y is fair. This

can be seen most easily be noting that the classes of the partition for both automata correspond to the tasks being processed (A_M has additional classes for the *receive* actions). Thus fairness for both automata is equivalent to saying that for any $t \in$ tasks(a) for some state a of a fair execution, an action *process*$_t$ will occur in that execution. Since h preserves the set of tasks, it follows x is fair iff y is fair. □

6.3.3 Correspondence between I/O Automata Actors and Agha's Model

We will henceforth consider only the simpler system A. It should be clear that, for A defined with respect to $\mathbf{A}$ (in other words having the same set of behaviors $\mathcal{B}$), there is a correspondence between A and $\mathbf{A}$. We now prove this.

Theorem 6.2 *There is a bijection h_1 between* states(A) *and* states($\mathbf{A}$). *There is a bijection h_3 between* steps(A) *and* steps($\mathbf{A}$) *preserving h_1. There is a bijection h_5 between* fair(A) *and* fair($\mathbf{A}$) *preserving h_3.*

Proof: This is proven in a series of steps.

1. There is a bijection h_1 between states(A) and states($\mathbf{A}$). Just define

$$h_1(a) = (\text{actors}(a), \text{tasks}(a))$$

. It is a bijection because any task in tasks(a) must be in a unique queue corresponding to its target, and all queues corresponding to mail addresses that do not belong to actors in actors(a) must be empty.

2. Let $E(a)$ be the set of actions enabled from state a of A and $E(h_1(a))$ be the set of tasks enabled from $h_1(a)$. Then there is a bijection h_2^a between $E(a)$ and $E(h_1(a))$. This follows directly since each *process*$_t$ action corresponds to the task t it processes, and exactly one *process*$_t$ action is enabled from a for each t enabled from $h_1(a)$.

3. There is a bijection h_3 between steps(A) and steps($\mathbf{A}$) preserving h_1. Just define

$$h_3((a, \pi, a')) = (h_1(a), h_2^a(\pi), h_1(a'))$$

.

4. There is a bijection h_4 between execs(A) and execs(**A**) preserving h_3. Follows directly from the previous step.

5. There is a bijection h_5 between fair(A) and fair(**A**) preserving h_3. Simply restrict the domain of h_4 to fair(A). Let $C(t)$ be the class of part(A) containing all $process_t$ actions. In a fair execution of A, if any action in $C(t)$ is enabled then some action of $C(t)$ will occur. In a fair execution of **A**, every task is processed. The equivalence follows from h_2^a.

□

From Theorem 6.2 it follows that A exactly specifies **A**.

6.4 Correctness Proofs

Now that **A** has been formally defined, both in terms of Agha's operation model and in terms of the I/O automaton A, one can prove correctness of actor programs either within Agha's model or the I/O automaton model. Both models assume that the behavior of each actor is given. Therefore if one wishes to prove correctness of an actor program written in some language such as SAL, one must first determine the behavior corresponding to this program. This is discussed in Section 5.2 of Agha's book, and will not be elaborated upon here.

Agha's model and I/O automata each have their own advantages. The advantage of Agha's model is that it is a very clean model of actors and only the essential details are included. However until now no proofs have been done in this model, and so no proof techniques have been developed. I/O automata, on the other hand, have been used for correctness proofs for some time, and many techniques have been developed. So it might be worthwhile for more complicated examples to translate them into I/O automata and use these techniques. However at the present time there are several complications due to Agha's model with severely inhibit doing this. These complications and possible solutions will be discussed at the end of the section.

We will prove correctness of the recursive factorial in Agha's model. The corresponding proof in I/O automata is essentially identical.

6.4.1 Recursive Factorial Revisited

Formal Behaviors

Agha formally defines the *Rec-Factorial* behavior as φ, where

$$\varphi(t,m,[k_1,k_2]) = \begin{cases} \langle\{(t.1,k_2,[1])\},\emptyset,(m,\varphi)\rangle & \text{if } k_1 = 0 \\ \langle\{(t.1,m,[k_1-1,t.2])\},\left\{(t.2,\psi_{k_2}^{k_1})\right\},(m,\varphi)\rangle & \text{otherwise} \end{cases}$$

and the *Rec-Customer* behavior is formally defined to be

$$\psi_{k_2}^{k_1}(t',m',[n]) = \langle\{(t'.1,k_2,[n*k_1])\},\emptyset,(m',\beta_\perp)\rangle$$

Here $\beta_\perp$ is the "bottom behavior," equivalent to an infinite sink (in other words a behvior that, upon processing a task, does nothing). Note that Agha contradicts an earlier statement (p. 33) that if a replacement behavior is not specified in the code (as it isn't in Figure 6-1) then the replacement behavior is the same the current behavior. He then says (p. 75) "It can be shown that in any actor system this newly created actor will receive at most one communication; thus the behavior of its replacement is actually irrelevant." However this is not necessarily true unless one assumes an actor with behavior *Rec-Customer* can only be created by an actor with behavior *Rec-Factorial*. In any case, we will adopt the convention that the replacement behavior is the same as the current behavior if no replacement behavior is explicitly indicated. Therefore the *Rec-Customer* behavior will be defined as

$$\psi_{k_2}^{k_1}(t',m',[n]) = \langle\{(t'.1,k_2,[n*k_1])\},\emptyset,(m',\psi_{k_2}^{k_1})\rangle$$

Note also that the tags and mail addresses of newly-created tasks and actors are overspecified, but this is not really a problem. Tags and mail addresses are only used to maintain uniqueness and to create other tags and mail addresses. So it does not matter if one specifies them concretely as was done for these examples.

Correctness Proof

Note that in Agha's model there is simply a single automaton **A** and that any state can be a start state; the same is true of the I/O automaton A. Therefore "proving correctness" simply means proving **A** (or A) has some desired property.

Let $x = a_0\pi_1a_1\ldots$ be a fair execution of **A**. Let n, i, i' be nonnegative integers such that $i \leq i'$; let m and u be mail addresses and let t be a tag. A correctness property for recursive factorial is: If $(m, \varphi) \in \text{actors}(a_i)$ and $(t, m, [n, u]) \in \text{tasks}(a_{i'})$, then there exists some $j > i'$ and some tag t' such that $(t', u, [n!]) \in \text{tasks}(a_j)$. Informally what is being said is that if a task is sent to a recursive factorial actor with mail address m requesting that it compute $n!$ and send the result to mail address u, then this actor will fulfill the request.

First it should be clear that a property of the rec-customer behavior is as follows. For nonnegative integers k, n, i, i' such that $i \leq i'$, mail addresses m, u, p and tag t, if $(m, \psi_p^k) \in$ actors(a_i) and $(t, u, [n]) \in \text{tasks}(a_{i'})$, then there exists some $j > i'$ and some tag t' such that $(t', u, [n * k]) \in \text{tasks}(a_j)$. This follows from the fact that (m, ψ_p^k) never changes behavior and from fairness.

Let $(m, \varphi) \in \text{actors}(a_i)$ and $(t, m, [n, u]) \in \text{tasks}(a_{i'})$ where $i \leq i'$. Note that $(m, \varphi) \in a_q$ for all $q \geq i$. The proof is by induction on n. If $n = 0$ then there must exist some $j > i'$ such that $\pi_j = (t, m, [0, u])$ due to fairness. It follows from φ that $(t.1, u, [1]) \in \text{tasks}(a_j)$.

If $n > 0$ then there must exist some $j > i'$ such that $\pi_j = (t, m, [n, u])$ due to fairness. It follows from φ that $(t.1, m, [n-1, t.2]) \in \text{tasks}(a_j)$ and $(t.2, \psi_u^n) \in \text{actors}(a_j)$. By the induction hypothesis there exists some $j' > j$ such that $(t', t.2, [(n-1)!]) \in \text{tasks}(a_{j'})$ for some tag t'. From the correctness property of ψ_u^n it follows there is some $j'' > j'$ such that $(t'', u, [n!]) \in \text{tasks}(a_{j''})$ for some tag t''. This completes the proof.

6.4.2 Comments

The proof of correctness for recursive factorial given above was not very complicated, although this is primarily because the algorithm, although it involves dynamic process creation, is inherently sequential. However for more complicated examples it may be very cumbersome to prove correctness. We examine some reasons for this.

The proof of recursive factorial was done at a low level, examining a particular execution. It would be nice to abstract away some of the details. In particular it can be seen that the proof has a sort of rely/guarantee flavor to it, and it seems proofs of actor programs using rely/guarantee functions would be a good idea. This is because actors, like I/O automata, must respond to any sequence of inputs. Therefore an actor will generally rely upon the

environment to not send it any bad inputs. Herein lies one problem: the lack of restriction on communications. Recall that in the language SAL there was a communication list describing the types of communications a behavior could handle. However this is not part of Agha's model. Of course one can prove that every communication sent is of the proper form, but this is an extra burden on the prover. Similarly acquaintance lists, indicating which mail addresses an actor may send a communication to, are absent in Agha's model. The acquaintance list in particular should be added to the model, as it can be used to define the topology of an actor system. The proof of recursive factorial would be much more complicated if the actors had been able to change behavior, because we would have to have shown that they did not change behavior before receiving the task of interest, and that in turn would have meant showing that no other actor could have sent them a task. Use of acquaintances would make such a proof much more tractable.

Another problem that hinders using rely/guarantee functions is the lack of modularity and abstraction. Several features of Agha's model contribute to this. One is that there is no real notion of composition and action hiding. One cannot compose two actors to be a single actor, since every actor must have a unique mail address. Agha does attempt to define a notion of composition for sets of actors in section 7.2.[illegible] of his book. In some sense this is similar to simply composing two or more I/O automata that represent actors. However he also attempts to define notions of input, output and external actions similar to those in I/O automata. Given a set of actors, he defines some subset to be *receptionists* for the set; they are the only ones which may receive communications from the environment. Other actors may be *external actors*; these represent actors outside the set and can be thought of as outputs for the set. However there are some problems with this approach. One is that the sets of receptionists and external actors of a set S of actors depend upon which other set S is composed with. This could be corrected by making the sets of receptionists and external actors fixed, and defining a notion of "compatibility" such as that for I/O automata to determine when two sets can be composed. Of course mail addresses can be renamed to make two sets compatible. Another problem is that, given a step (a, π, a') of $\mathbf{A}$ along with the receptionists and external actors of some set of actors S in a, it is not determined what the receptionists and external actors of S are in a'. This could be fixed by defining a function on communications that returns the mail

addresses "communicated" in that message; a notion of acquaintances would be helpful here too. So although Agha shows that you can compose two sets of actors, without some changes or additions to his model it seems that this notion is not useful.

When doing a proof of a dynamic system it is helpful to use as much static information as possible. All useful algorithms have some sort of structure which can be exploited. In dynamic systems this structure is typically a process creation tree; this was used particularly in the proof of Olympic Torch. There are process creation trees in recursive factorial as well, but these cannot be determined statically, for example by examining the mail addresses of the actors. This in turn is because of the naming scheme for mail addresses and tags, in which mail addresses of newly created actors are based upon the tag of the task that is processed to create them rather the mail address of the actor creating them. It would be useful to have a different naming system in which mail addresses of actors are prefixes of the mail addresses of their children, so that this information could be determined statically. This would allow one to compose these processes and consider them as a unit. As it is now, one can only determine the process creation tree of an actor by examining each individual execution.

There's no reason why correctness proofs of actor programs should be more difficult than those for dynamic systems written in I/O automata. Adding features to Agha's model of actors to allow composition, abstraction, and static evaluation of some properties would make such proofs considerably more tractable.

Chapter 7

Conclusions

It has been shown how process creation can be handled in the I/O automata model, and proofs of correctness have been given for two algorithms which use process creation. It can be seen from these examples that process creation is itself not a complication in correctness proofs; rather it is the structure of the algorithm which determines how complicated the proof is. Olympic Torch uses a considerable amount of process creation and a changing topology, yet the basic structure of the algorithm is fairly simple and therefore the proof can be as well.

Rely/guarantee functions have been defined and used to give modular, intuitive correctness proofs for both Set Partition and Olympic Torch. Note that the former uses the technique of successive refinement, whereas for the latter one simply composes the functions, using safety properties to simplify this composition. Originally the proof of Olympic Torch had been done using successive refinement as well, but this wasn't as clear as doing the composition directly. The moral seems to be that no proof technique is the best for all examples; the proof should be based upon the algorithm. However rely/guarantee functions do seem to be a good way to organize proofs in general, and more work should be done to see how they can be applied.

Finally Actors, a model for dynamic systems, has been defined using I/O automata. In doing so a number of errors in Agha's model of Actors have been fixed. These points probably would have been discovered if a correctness proof had been attempted in the model, though. Actors, with its emphasis on flexibility, does not seem to be a good model in which to do correctness proofs of general distributed algorithms. However, with some work, it should be a good model in which to do correctness proofs of algorithms written in an Actor language. A good direction for

future research is to enhance the Actor model, adding features for composition and abstraction, so that one can specify and prove correctness more easily in it.

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[AL90] Martín Abadi and Leslie Lamport. Composing specifications. 1990. Digital Equipment Corporation.

[Bar85] Howard Barringer. *A Survey of Verification Techniques for Parallel Programs. Lecture Notes in Computer Science 191*, Springer-Verlag, Berlin, 1985.

[Cli81] William Douglas Clinger. *Foundations of Actor Semantics.* Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.

[dC89] Dennis de Champeaux. Verification of some parallel algorithms. In *Proceedings 1989 Pacific Northwest Software Quality Conference*, pages 149–169, 1989. Also available as Hewlett-Packard Laboratories Technical Report STL-89-12.

[Gol90] Kenneth J. Goldman. *Distributed Algorithm Simulation using Input/Output Automata.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1990. (in progress).

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In *Proc. IFIP 83*, pages 321–332, North Holland Publishing Company, 1983.

[LG81] Gary Marc Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, June 1981.

[LT87] Nancy A. Lynch and Mark R. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, April 1987.

[LT88] Nancy A. Lynch and Mark R. Tuttle. *An Introduction to Input/Output Automata.* Technical Memo MIT/LCS/TM-373, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, November 1988.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.

[MCS82] Jayadev Misra, K. Mani Chandy, and Todd Smith. Proving safety and liveness of communicating processes with examples. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 201–208, ACM, August 1982.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, August 1976.

[Sta84] Eugene W. Stark. *Foundations of a Theory of Specification for Distributed Systems.* Technical Report MIT/LCS/TR-342, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, August 1984.

[Yon77] Akinori Yonezawa. *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics.* Technical Report MIT/LCS/TR-191, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, December 1977.