Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Verification of a distributed summation algorithm

F.W. Vaandrager

Computer Science/Department of Software Technology

CS-R9505 1995

Report CS-R9505
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Verification of a Distributed Summation Algorithm

Frits W. Vaandrager
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`fritsv@cwi.nl`
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract

A correctness proof of a variant of Segall's Propagation of Information with Feedback protocol is presented. The proof, which is carried out within the I/O automata model of Lynch and Tuttle, is standard except for the use of a prophecy variable. The aim of this paper is to show that, unlike what has been suggested in the literature, assertional methods based on invariant reasoning support an intuitive way to think about and understand this algorithm.

AMS Subject Classification (1991): 68M10, 68Q05, 68Q22, 68Q25, 68Q60.
CR Subject Classification (1991): C.2.2, D.2.4, F.1.1, F.3.1.
Keywords & Phrases: Distributed network protocols, Protocol verification, I/O automata, Invariants, History variables, Prophecy variables, Refinements.
Notes: This paper is dedicated to P.C. Baayen who, like the root node in Segall's algorithm, initiated a cascade of mathematical activity. A preliminary version appeared in: K.R. Apt, A. Schrijver, N.M. Temme, editors, *From Universal Morphisms to Megabytes — a Baayen Space Odyssey*, pages 593–608, CWI, Amsterdam, December 1994. The author was supported by Esprit BRA 7166 CONCUR 2.

1. Introduction

Reasoning about distributed algorithms appears to be intrinsically difficult and will probably always require a great deal of ingenuity. Nevertheless, research on formal verification has provided a whole range of well-established concepts and techniques that may help us to tackle problems in this area. It seems that by now the basic principles for reasoning about distributed algorithms have been discovered and that the main issue that remains is the problem of scale: we know how to analyze small algorithms but are still lacking methods and tools to manage the complexity of the the bigger ones.

Not everybody agrees with this view, however, and frequently one can hear claims that existing approaches cannot deal in a 'natural' way with certain types of distributed algorithms. A new approach is then proposed to address this problem. A recent example of this is a paper by Chou [4], who offers a rather pessimistic view on the state-of-the-art in formal verification:

> At present, reasoning about distributed algorithms is still an *ad hoc*, trial-and-error process that needs a great deal of ingenuity. What is lacking is a practical

method that supports, on the one hand, an *intuitive* way to think about and understand distributed algorithms and, on the other hand, a *formal* technique for reasoning about distributed algorithms using that intuitive understanding.

In his paper, Chou proposes an extension of the assertional methods of [2, 7, 8, 9, 10, 12, 15], and argues that this extension allows for a more direct formalization of intuitive, operational reasoning about distributed algorithms. To illustrate his method, Chou discusses a variant of Segall's PIF (Propagation of Information with Feedback) protocol [19]. A complex and messy proof of this algorithm using existing methods is contrasted with a slightly simpler but definitely more structured proof based on the new method.

Is the process of using assertional methods based on invariant reasoning ad hoc? Personally, I believe it is not. On the contrary, I find that these methods provide significant guidance and structure to verifications. After one has described both the algorithm and its specification as abstract programs, it is usually not so difficult to come up with a first guess of a simulation relation from the state space of the algorithm to the state space of the specification. In order to state this simulation, it is sometimes necessary to add auxiliary history and prophecy variables to the low-level program. By just starting to prove that the guessed simulation relation is indeed a simulation, i.e., that for each execution of the low-level program there exists a corresponding execution of the high-level program, one discovers the need for certain invariants, properties that are valid for all reachable states of the programs. To state these invariant properties it is sometimes convenient or even necessary to introduce auxiliary state variables. Frequently one also has to prove other auxiliary invariants first. The existence of a simulation relation guarantees that the algorithm is safe with respect to the specification: all the finite behaviors of the algorithm are allowed by the specification. The concepts of invariants, history and prophecy variables, and simulation relations are so powerful that in most cases they allow one to formalize the intuitive reasoning about safety properties of distributed algorithms. When a simulation (and thereby safety) has been established, this simulation often provides guidance in the subsequent proof that the algorithm satisfies the required liveness properties: typically one proves that the simulation relates each fair execution of the low-level program to a fair execution of the high-level program. Here modalities from temporal logic such as "eventually" and "leads to" often make it quite easy to formalize intuitions about the liveness properties of the algorithm.

As an illustration of the use of existing assertional methods, I present in this paper a verification within the I/O automata model [12, 13] of the algorithm discussed by Chou [4]. Altogether, it took me about two hours to come up with a sketch of the proof (during a train ride from Leiden to Eindhoven), and about three weeks to work it out, polish it, and write this paper. The proof is routine, except for a few nice invariants and the use of a prophecy variable. Unlike history variables, which date back to the sixties [11], prophecy variables have been introduced only recently [1], and there are not that many examples of their use. My proof is not particularly short, but it does formalize in a direct way my own intuitions about the behavior of this algorithm. It might very well be the case that for more complex distributed algorithms new methods, such as the one of Chou [4], will pay off and lead to shorter proofs that are closer to intuition. This paper shows that invariant based assertional methods still work very well for a variant of Segall's PIF protocol.

The structure of this paper is as follows. Section 2 describes the algorithm formally as an I/O automaton. Section 3 presents the correctness criterion and the proof that the algorithm meets this criterion. Finally, Section 4 contains some conclusing remarks. Appendix A gives a brief account of those parts of I/O automata theory that are used in this paper.

2. Description of the Algorithm

We consider a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is a nonempty, finite collection of nodes and $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a collection of links. We assume that graph $\mathbf{G}$ is undirected in the sense that $(v, w) \in \mathbf{E} \Leftrightarrow (w, v) \in \mathbf{E}$, and connected. To each node v in the graph a value $\mathsf{weight}(v)$ is associated, taken from some set $\mathbf{M}$. We assume that $\mathbf{M}$ contains an element 0 and that there is a binary operator $+$ on $\mathbf{M}$, such that $(\mathbf{M}, +, 0)$ is an Abelian monoid.[1]

Nodes of $\mathbf{G}$ represent autonomous processors and links represent communication channels via which these processors can send messages to each other. We assume that the communication channels are reliable and that messages are received in the same order as they are sent. We discuss a simple distributed algorithm to compute the sum of the weights of all the nodes in the network. The algorithm is a minor[2] rephrasing of an algorithm described by Chou [4], which in turn is a variant of Segall's PIF (Propagation of Information with Feedback) protocol [19].

The only messages that are required by the algorithm are elements from $\mathbf{M}$. A node in the network enters the protocol when it receives a first message from one of its neighbors. Initially, the communication channels for all the links are empty, except the channel associated to the link e_0 from a fixed root node v_0 to itself, which contains a single message.[3] When an arbitrary node v receives a first message, it marks the node w from which this message was received. It then sends a 0 message to all its neighbors, except w. Upon receiving subsequent messages, the values of these messages are added to the weight of v. As soon as, for a non-root node, the total number of received messages equals the total number of neighbors, the value that has been computed is sent back to the node from which the first message was received. When, for root node v_0, the total number of received messages equals the total number of neighbors, the value that has been computed by v_0 is produced as the final outcome of the algorithm.

In Figure 1, the algorithm is specified as an I/O automaton *DSum* using the standard precondition/effect notation [12, 13, 6]. A minor subtlety is the occurrence of the variable v in the definition of the step relation, which is neither a state variable nor a formal parameter of the actions. Semantically, the meaning of v is determined by an implicit existential quantification: an action a is enabled in a state s if there exists a valuation ξ of all the variables (including v) that agrees with s on the state variables and with a on the parameters of the actions, such that the precondition of a holds under ξ. If action a is enabled in s under ξ then the effect part of a and ξ determine the resulting state s'.

For each link $e{=}(v, w)$, the source v is denoted $\mathsf{source}(e)$, the target w is denoted $\mathsf{target}(e)$,

[1]So, for all $m, m', m'' \in \mathbf{M}$, $m + m' = m' + m$, $m + (m' + m'') = (m + m') + m''$ and $m + 0 = 0 + m = m$.

[2]The unit element 0 of the monoid is used where Chou [4] uses a special `Start` message.

[3]The assumption that $\mathsf{e}_0 = (\mathsf{v}_0, \mathsf{v}_0) \in \mathbf{E}$ is not required, but allows for a more uniform description of the algorithm for each node.

```
Internal:  $MSG$
           $REPORT$
Output:    $RESULT$

State Variables: $busy : \mathbf{V} \rightarrow \mathbf{Bool}$
                 $par : \mathbf{V} \rightarrow \mathbf{E}$
                 $total : \mathbf{V} \rightarrow \mathbf{M}$
                 $cnt : \mathbf{V} \rightarrow \mathbf{Int}$
                 $mq : \mathbf{E} \rightarrow \mathbf{M}^*$

Init: $\wedge\ \forall v : \neg busy[v]$
      $\wedge\ \forall e : mq[e] = \text{if } e{=}\mathsf{e}_0 \text{ then } \mathsf{append}(0, \mathsf{empty}) \text{ else } \mathsf{empty}$

$MSG(e : \mathbf{E}, m : \mathbf{M})$
    Precondition:
        $v = \mathsf{target}(e) \wedge m = \mathsf{head}(mq[e])$
    Effect:
        $mq[e] := \mathsf{tail}(mq[e])$
        if $\neg busy[v]$ then $busy[v] := \mathsf{true}$
                               $par[v] := e$
                               $total[v] := \mathsf{weight}(v)$
                               $cnt[v] := \mathsf{size}(\mathsf{to}(v)) - 1$
                               for $f \in \mathsf{from}(v)/\{e^{-1}\}$ do $mq[f] := \mathsf{append}(0, mq[f])$
                          else $total[v] := total[v] + m$
                               $cnt[v] := cnt[v] - 1$

$REPORT(e : \mathbf{E}, m : \mathbf{M})$
    Precondition:
        $v = \mathsf{source}(e) \neq \mathsf{v}_0 \wedge busy[v] \wedge cnt[v] = 0 \wedge e^{-1} = par[v] \wedge m = total[v]$
    Effect:
        $busy[v] := \mathsf{false}$
        $mq[e] := \mathsf{append}(m, mq[e])$

$RESULT(m : \mathbf{M})$
    Precondition:
        $busy[\mathsf{v}_0] \wedge cnt[\mathsf{v}_0] = 0 \wedge m = total[\mathsf{v}_0]$
    Effect:
        $busy[\mathsf{v}_0] := \mathsf{false}$
```

Figure 1: I/O automaton *DSum*.

and the reverse link (w, v) is denoted e^{-1}. For each node v, $\mathsf{from}(v)$ gives the set of links with source v and $\mathsf{to}(v)$ gives the set of links with target v, so $e \in \mathsf{from}(v) \Leftrightarrow \mathsf{source}(e) = v$ and $e \in \mathsf{to}(v) \Leftrightarrow \mathsf{target}(e) = v$. All the other data types and operation symbols used in the specification have the obvious meaning. The states of *DSum* are interpretations of five state variables in their domains. Four of these variables represent the values of program variables at each node:

- *busy* tells for each node whether or not it is currently participating in the protocol; initially $busy[v] = \mathsf{false}$ for each v;
- *par* is used to remember the link via which a node has been activated;
- *total* records the sum of the values seen by a node during a run of the protocol;
- *cnt* gives the number of values that a node still wants to see before it will terminate.

The fifth state variable *mq* represents the contents of the message queue for each link. Initially, $mq[e]$ is empty for each link e except $\mathsf{e_0}$.

I/O automaton *DSum* has three (parametrized) actions: (1) *MSG*, which describes the receipt and processing of a message, (2) *REPORT*, by which a non root node sends the final value that it has computed to its parent, and (3) *RESULT*, which is used by the root node to deliver the final result of the computation. The partition of *DSum* contains an equivalence class B_v for each node v, which gives all the actions in which node v participates:

$$B_{\mathsf{v_0}} \triangleq \{MSG(e, m) \mid e \in \mathsf{to}(\mathsf{v_0}),\ m \in \mathbf{M}\} \cup \{RESULT(m) \mid m \in \mathbf{M}\}$$

and, for $v \neq \mathsf{v_0}$,

$$B_v \triangleq \{MSG(e, m) \mid e \in \mathsf{to}(v),\ m \in \mathbf{M}\} \cup \{REPORT(e, m) \mid e \in \mathsf{from}(v),\ m \in \mathbf{M}\}$$

Actually, since it will turn out that *DSum* only has finite executions, it does not matter how we define the partition of *DSum*. The above definition seems to be the most natural, since it reflects the intuition that each node in the network represents an autonomous processor.

3. Correctness Proof

3.1 Correctness Criterion

The correctness property that we want to establish is that the fair traces of *DSum* are contained in those of the I/O automaton S of Figure 2. I/O automaton S is extremely simple. It has only two states: an initial state where $done = \mathsf{false}$ and a final state where $done = \mathsf{true}$. There is one step, which starts in the initial state, has label $RESULT(\sum_{v \in \mathbf{V}} \mathsf{weight}(v))$, and ends in the final state. Finally, $part(S)$ contains a single equivalence class $\{RESULT(m) \mid m \in \mathbf{M}\}$.

We will prove $traces(DSum) \subseteq traces(S)$ using a standard recipe of Abadi and Lamport [1]: first we establish a history relation from *DSum* to an I/O automaton $DSum^h$, then a prophecy relation from $DSum^h$ to an I/O automaton $DSum^{hp}$, and finally a refinement from $DSum^{hp}$ to S. The fact that $traces(DSum) \subseteq traces(S)$ does not guarantee that $fairtraces(DSum) \subseteq fairtraces(S)$. In order to prove this, we will show that *DSum* has no infinite sequence of consecutive internal actions and cannot get into a state of deadlock before an output step has been performed.

```
Output:   $RESULT$

State Variables: $done$ : Bool

Init: $\neg done$

$RESULT(m : \mathbf{M})$
    Precondition:
        $\neg done \wedge m = \sum_{v \in \mathbf{V}} \mathsf{weight}(v)$
    Effect:
        $done := \mathsf{true}$
```

Figure 2: I/O automaton S.

3.2 Adding a History Variable

As observed by Segall [19], a crucial property of the PIF protocol is that in each maximal execution exactly one message travels on each link. As a first step towards the proof of this property, we will establish that in each execution of *DSum* at most one message travels on each link. In order to state this formally as an invariant, we add a variable *rcvd* to automaton *DSum* that records for each link e how many messages have been received on e. This variable is similar to the variable N that Segall [19] uses in his presentation of PIF to mark the receipt of a message over a link. Figure 3 describes the automaton $DSum^h$ obtained in this way. Boxes highlight the places where $DSum^h$ differs from *DSum*. Variable *rcvd* is an auxiliary/history variable in the sense of Owicki and Gries [18] because it does not occur in conditions nor at the right-hand-side of assignments to other variables. Clearly, adding *rcvd* does not change the behavior of automaton *DSum*. This can be formalized via the following trivial result (Here a strong history relation is a relation on states whose inverse is a functional strong bisimulation. See Appendix A.3 for the definition.).

Theorem 1 *The inverse of the projection function that maps states from $DSum^h$ to states of DSum is a strong history relation from DSum to $DSum^h$.*

We will use a state function $Sent(e)$ to denote the number of messages sent over a link e, and a state function $Rcvd(v)$ to denote the number of messages received by a node v. Formally, these functions are defined by:

$$\begin{aligned} Sent(e) &\triangleq rcvd[e] + \mathsf{len}(mq[e]) \\ Rcvd(v) &\triangleq \textstyle\sum_{e \in \mathsf{to}(v)} rcvd[e] \end{aligned}$$

Invariant I below gives some basic sanity properties involving $rcvd[e]$ and $Rcvd(v)$: at any time the number of messages received from a link is nonnegative, if a node is busy then it has received at least one message, and as soon as at least one message has been received by a node, a message has been received over the parent link, which points towards that node.

```
Internal:  $MSG$
           $REPORT$
Output:    $RESULT$

State Variables: $busy : \mathbf{V} \rightarrow \mathbf{Bool}$
                 $par : \mathbf{V} \rightarrow \mathbf{E}$
                 $total : \mathbf{V} \rightarrow \mathbf{M}$
                 $cnt : \mathbf{V} \rightarrow \mathbf{Int}$
                 $mq : \mathbf{E} \rightarrow \mathbf{M}^*$
                 $\boxed{rcvd : \mathbf{E} \rightarrow \mathbf{Int}}$

Init: $\wedge\ \forall v : \neg busy[v]$
      $\wedge\ \forall e : mq[e] = \text{if } e{=}\mathsf{e}_0 \text{ then append}(0, \text{empty}) \text{ else empty}$
      $\boxed{\wedge\ \forall e : rcvd[e] = 0}$

$MSG(e : \mathbf{E}, m : \mathbf{M})$
    Precondition:
        $v = \text{target}(e) \wedge m = \text{head}(mq[e])$
    Effect:
        $mq[e] := \text{tail}(mq[e])$
        $\boxed{rcvd[e] := rcvd[e] + 1}$
        if $\neg busy[v]$ then $busy[v] := \text{true}$
                               $par[v] := e$
                               $total[v] := \text{weight}(v)$
                               $cnt[v] := \text{size}(\text{to}(v)) - 1$
                               for $f \in \text{from}(v)/\{e^{-1}\}$ do $mq[f] := \text{append}(0, mq[f])$
                          else $total[v] := total[v] + m$
                               $cnt[v] := cnt[v] - 1$

$REPORT(e : \mathbf{E}, m : \mathbf{M})$
    Precondition:
        $v = \text{source}(e) \neq \mathsf{v}_0 \wedge busy[v] \wedge cnt[v] = 0 \wedge e^{-1} = par[v] \wedge m = total[v]$
    Effect:
        $busy[v] := \text{false}$
        $mq[e] := \text{append}(m, mq[e])$

$RESULT(m : \mathbf{M})$
    Precondition:
        $busy[\mathsf{v}_0] \wedge cnt[\mathsf{v}_0] = 0 \wedge m = total[\mathsf{v}_0]$
    Effect:
        $busy[\mathsf{v}_0] := \text{false}$
```

Figure 3: I/O automaton $DSum^h$ obtained from $DSum$ by adding history variable $rcvd$.

Lemma 2 *Let I be the conjunction, for all v and e, of the following properties:*

$$\begin{aligned} I_1(e) &\triangleq rcvd[e] \geq 0 \\ I_2(v) &\triangleq busy[v] \rightarrow Rcvd(v) > 0 \\ I_3(v) &\triangleq Rcvd(v) > 0 \rightarrow par[v] \in \mathsf{to}(v) \wedge rcvd[par[v]] > 0 \end{aligned}$$

Then I holds for all reachable states of $DSum^h$.

The real work starts with the proof of the next invariant I', which is the conjunction, for all v, of the following formulas:

$$\begin{aligned} I_4(v) &\triangleq Init(v) \vee busy[v] \vee Done(v) \\ I_5 &\triangleq Init(\mathsf{v_0}) \rightarrow \mathbf{Init}(DSum^h) \\ I_6(v) &\triangleq v \neq \mathsf{v_0} \wedge Init(v) \rightarrow \forall e \in \mathsf{from}(v) : Sent(e) = 0 \\ I_7(v) &\triangleq \neg Init(v) \rightarrow \forall e \in \mathsf{from}(v)/\{par[v]^{-1}\} : Sent(e) = 1 \\ I_8 &\triangleq \neg Init(\mathsf{v_0}) \rightarrow par(\mathsf{v_0}) = \mathsf{e_0} \wedge Sent(\mathsf{e_0}) = 1 \\ I_9(v) &\triangleq v \neq \mathsf{v_0} \wedge busy[v] \rightarrow Sent(par[v]^{-1}) = 0 \\ I_{10}(v) &\triangleq v \neq \mathsf{v_0} \wedge Done(v) \rightarrow Sent(par[v]^{-1}) = 1 \\ I_{11}(v) &\triangleq \neg Init(v) \rightarrow cnt[v] + Rcvd(v) = \mathsf{size}(\mathsf{to}(v)) \end{aligned}$$

in which the following state functions are used:

$$\begin{aligned} Init(v) &\triangleq \forall e \in \mathsf{to}(v) : rcvd[e] = 0 \\ Done(v) &\triangleq \neg busy[v] \wedge \forall e \in \mathsf{to}(v) : rcvd[e] = 1 \end{aligned}$$

Even though at first sight formula I' may look complicated, it is easy to give intuition for it. As long as a node v has not received any message, it does not participate is the protocol and is in state $Init(v)$. Upon arrival of a first message, the node changes status and moves to $busy[v]$. The node remains in this state until it has received a message from all its neighbors, then performs a *REPORT* or *RESULT* action, and moves to its final state $Done(v)$. The following "mutual exclusion" property is a logical consequence of invariant I_2 and the definition of state functions *Init* and *Done*, and therefore holds for all reachable states of $DSum^h$:

$$ME(v) \triangleq \neg(Init(v) \wedge busy[v]) \wedge \neg(Init(v) \wedge Done(v)) \wedge \neg(busy[v] \wedge Done(v))$$

Together with formula $I_4(v)$, $ME(v)$ says that in any reachable state each node is in exactly one of the three states $Init(v)$, $busy[v]$ or $Done(v)$. Formulas I_5-I_{10} specify, for each node v, for each of the three possible states of v, and for each outgoing link of v, how many messages have been sent over that link. And since this number is always either 0 or 1, this implies that during each execution at most one message can be sent over each link (formula C_1 below). In order to make the induction work, a final conjunct I_{11} is needed in I' that says that, except for the initial state of v, $cnt[v]$ gives the total number of links over which no message has yet been received by v. In the routine proof that I' is an invariant, it is convenient to use the logical consequences C_1-C_3 of $I \wedge I'$ that are stated in Lemma 3. Properties C_4-C_7 of Lemma 3 are also logical consequences of $I \wedge I'$, and will play a role later on in this paper.

Lemma 3 *For all v and e, the following formulas are logical consequences of $I \wedge I'$ and the definitions of the state functions:*

$$\begin{aligned}
C_1(e) &\triangleq Sent(e) \leq 1\\
C_2(v) &\triangleq (\forall e \in \mathsf{to}(v) : rcvd[e] = 1) \ \leftrightarrow\ Rcvd(v) = \mathsf{size}(\mathsf{to}(v))\\
C_3(e) &\triangleq mq[e] \neq \mathsf{empty} \wedge \neg busy[\mathsf{target}(e)] \ \rightarrow\ Init(\mathsf{target}(e))\\
C_4(e) &\triangleq e \neq \mathsf{e}_0 \wedge mq[e] \neq \mathsf{empty} \ \rightarrow\ \neg Init(\mathsf{source}(e))\\
C_5(v) &\triangleq \neg Init(v) \ \rightarrow\ \neg Init(\mathsf{source}(par[v]))\\
C_6(e) &\triangleq Init(\mathsf{target}(e)) \wedge mq[e] = \mathsf{empty} \ \rightarrow\ Init(\mathsf{source}(e))\\
C_7(v) &\triangleq v \neq \mathsf{v}_0 \wedge \neg Init(v) \wedge Sent(par[v]^{-1}) = 1 \ \rightarrow\ Done(v)
\end{aligned}$$

Lemma 4 *Property I' holds for all reachable states of $DSum^h$.*

3.3 Adding a Prophecy Variable

Intuitively, in the first phase of the algorithm a spanning tree is constructed with root v_0, and this spanning tree is used to accumulate values in the second phase. When the algorithm starts, it not clear how the spanning tree is going to look like and in fact any spanning tree is still possible. While the algorithm proceeds, the spanning tree is constructed step by step. The choice whether an arbitrary link will be part of the spanning tree depends on the relative speeds of the processors, and is entirely nondeterministic. Such unpredictable, nondeterministic behavior is typical for distributed computation but often complicates analysis.

Fortunately, the concept of a *prophecy variable* of Abadi and Lamport [1] allows us to reduce the nondeterminism of the algorithm or, more precisely, to push nondeterminism backwards to the initial state. We add to $DSum^h$ a new variable *tree*, which records an initial guess of the spanning tree and enforces (as a self-fulfilling prophecy) that the actual spanning tree that is constructed during execution is equal to this initial guess. Figure 4 describes the automaton $DSum^{hp}$ obtained in this way. Boxes highlight the places where $DSum^{hp}$ differs from $DSum^h$. In Figure 4, tree is a function that tells for each set of links whether or not it is a tree. More formally, for $E \subseteq \mathbf{E}$ and $V = \{\mathsf{source}(e), \mathsf{target}(e) \mid e \in E\}$, $\mathsf{tree}(E) = \mathsf{true}$ iff either $E = \emptyset$ or there exists a node $v \in V$ such that for all $v' \in V$ there is a unique path of links in E leading from v to v'.

In order to show that *tree* is a prophecy variable in the sense of [1, 14], we establish a prophecy relation from $DSum^h$ to $DSum^{hp}$ (see Appendix A.3 for the definition). For this, we need two trivial invariants and two further lemmas.

Lemma 5 *For all reachable states of $DSum^{hp}$ and for all v:*

$$T_1(v) \ \triangleq\ \neg Init(v) \ \rightarrow\ par[v] = tree[v]$$

Lemma 6 *For all reachable states of $DSum^{hp}$:*

$$T_2 \ \triangleq\ tree[\mathsf{v}_0] = \mathsf{e}_0 \wedge (\forall v : v = \mathsf{target}(tree[v])) \wedge \mathsf{tree}(\{tree[v] \mid v \in \mathbf{V}/\{\mathsf{v}_0\}\})$$

Lemma 7 *Define $T \triangleq \forall v : T_1(v) \wedge T_2$ and let π be the projection function that maps states of $DSum^{hp}$ to states of $DSum^h$. Suppose a is an action and u and u' are states of $DSum^{hp}$ such that*

1. *$u.tree = u'.tree$,*
2. *$u' \models T$,*

```
Internal:   $MSG$
            $REPORT$
Output:     $RESULT$

State Variables: $busy : \mathbf{V} \to \mathbf{Bool}$
                 $par : \mathbf{V} \to \mathbf{E}$
                 $total : \mathbf{V} \to \mathbf{M}$
                 $cnt : \mathbf{V} \to \mathbf{Int}$
                 $mq : \mathbf{E} \to \mathbf{M}^*$
                 $rcvd : \mathbf{E} \to \mathbf{Int}$
                 $\boxed{tree : \mathbf{V} \to \mathbf{E}}$

Init: $\wedge\ \forall v : \neg busy[v]$
      $\wedge\ \forall e : mq[e] = \mathsf{if}\ e{=}\mathsf{e_0}\ \mathsf{then\ append}(0, \mathsf{empty})\ \mathsf{else\ empty}$
      $\wedge\ \forall e : rcvd[e] = 0$
      $\boxed{\wedge\ tree[\mathsf{v_0}] = \mathsf{e_0} \wedge (\forall v : v = \mathsf{target}(tree[v])) \wedge \mathsf{tree}(\{tree[v] \mid v \in \mathbf{V}/\{\mathsf{v_0}\}\})}$

$MSG(e : \mathbf{E}, m : \mathbf{M})$
  Precondition:
    $v = \mathsf{target}(e) \wedge m = \mathsf{head}(mq[e])\ \boxed{\wedge\ (busy[v] \vee e = tree[v])}$
  Effect:
    $mq[e] := \mathsf{tail}(mq[e])$
    $rcvd[e] := rcvd[e] + 1$
    if $\neg busy[v]$ then $busy[v] := \mathsf{true}$
                           $par[v] := e$
                           $total[v] := \mathsf{weight}(v)$
                           $cnt[v] := \mathsf{size}(\mathsf{to}(v)) - 1$
                           for $f \in \mathsf{from}(v)/\{e^{-1}\}$ do $mq[f] := \mathsf{append}(0, mq[f])$
                      else $total[v] := total[v] + m$
                           $cnt[v] := cnt[v] - 1$

$REPORT(e : \mathbf{E}, m : \mathbf{M})$
  Precondition:
    $v = \mathsf{source}(e) \neq \mathsf{v_0} \wedge busy[v] \wedge cnt[v] = 0 \wedge e^{-1} = par[v] \wedge m = total[v]$
  Effect:
    $busy[v] := \mathsf{false}$
    $mq[e] := \mathsf{append}(m, mq[e])$

$RESULT(m : \mathbf{M})$
  Precondition:
    $busy[\mathsf{v_0}] \wedge cnt[\mathsf{v_0}] = 0 \wedge m = total[\mathsf{v_0}]$
  Effect:
    $busy[\mathsf{v_0}] := \mathsf{false}$
```

Figure 4: I/O automaton $DSum^{hp}$ obtained from $DSum^h$ by adding prophecy variable *tree*.

3. $\pi(u) \xrightarrow{a} \pi(u')$ *is a step of* $DSum^h$*,*

4. $\pi(u)$ *is reachable.*

Then $u \xrightarrow{a} u'$ *is a step of* $DSum^{hp}$ *and* $u \models T$*.*

Proof: Since $u.tree = u'.tree$ and $u' \models T_2$, also $u \models T_2$. In order to show the remaining properties, we distinguish between three cases.
(1) $u \models \neg busy[v]$ and $a = MSG(e,m)$, for some v, e and m with $v = \mathsf{target}(e)$.
Since $\pi(u) \xrightarrow{a} \pi(u')$, it follows by C_3 that $\pi(u) \models Init(v)$ and therefore $u \models T_1(v)$. Because $u' \models \neg Init(v)$ and $u' \models T_1(v)$, $e = \pi(u').par[v] = u'.par[v] = u'.tree[v] = u.tree[v]$. Thus $u \xrightarrow{a} u'$. Clearly, $u \models T_1(w)$ for $w \neq v$, because $u' \models T_1(w)$ and because a does not change the relevant variables.
(2) $u \models busy[v]$ and $a = MSG(e,m)$, for some v, e and m with $v = \mathsf{target}(e)$.
Then $u \xrightarrow{a} u'$. Also $\pi(u).par[v] = \pi(u').par[v]$ and $\pi(u') \models busy[v]$. By the mutual exclusion property $ME(v)$, $u' \models \neg Init(v)$. Because $u' \models T_1(v)$, $u'.par[v] = u'.tree[v]$. Hence

$$u.par[v] = \pi(u).par[v] = \pi(u').par[v] = u'.par[v] = u'.tree[v] = u.tree[v].$$

This implies $u \models T_1(v)$. Finally we observe that $u \models T_1(w)$ for $w \neq v$, because $u' \models T_1(v)$ and because a does not change the relevant variables.
(3) For all e and m, $a \neq MSG(e,m)$.
Then $u \xrightarrow{a} u'$. Also, $u \models \forall v : T_1(v)$ because $u' \models \forall v : T_1(v)$ and because a does not change any of the relevant variables. □

Lemma 8 *Suppose* u *is a state of* $DSum^{hp}$ *such that* $\pi(u)$ *is reachable and* $u \models T$*. Then* u *is reachable.*

Proof: Let $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ be an execution of $DSum^h$ that ends in $\pi(u)$. Let, for $0 \leq i \leq n$, u_i be the state of $DSum^{hp}$ defined by $\pi(u_i) = s_i$ and $u_i.tree = u.tree$. Repeated application of Lemma 7 now gives that $DSum^{hp}$ has steps $u_0 \xrightarrow{a_1} u_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} u_n = u$ and, for all i, $u_i \models T$. Since $\pi(u_0)$ is a start state of $DSum^h$ and $u_0 \models T_2$, u_0 is a start state of $DSum^{hp}$, and thus u is reachable. □

Theorem 9 *The inverse of projection function* π *is a strong image-finite prophecy relation from* $DSum^h$ *to* $DSum^{hp}$*.*

Proof: Mapping π is trivially a strong refinement from $DSum^{hp}$ to $DSum^h$. Note that as a direct corollary of this fact, all invariants of $DSum^h$ are also invariants of $DSum^{hp}$. Since the domain of variable *tree* is finite, π^{-1} is image-finite. We prove that π^{-1} satisfies the three conditions of a backward simulation (condition (b) in the strong sense).

For condition (a), suppose that s is a start state of $DSum^h$ and u is a reachable state of $DSum^{hp}$ with $\pi(u) = s$. By Lemma 6, $u \models T_2$. Hence u is a start state of $DSum^{hp}$.

For condition (b), suppose that $s \xrightarrow{a} s'$ is a step of $DSum^h$, $\pi(u') = s'$ and s and u' are reachable. Let u be the state of $DSum^{hp}$ defined by $\pi(u) = s$ and $u.tree = u'.tree$. Since u' is reachable it follows by Lemmas 5 and 6 that $u' \models T$. Application of Lemma 7 now gives that $DSum^{hp}$ has a step $u \xrightarrow{a} u'$ and $u \models T$. Lemma 8 implies that u is reachable.

For condition (c), suppose that s is a reachable state of $DSum^h$. Let α be an execution of $DSum^h$ that ends in s. By induction on the number of steps in α we prove that there exists a reachable state of $DSum^{hp}$ that is mapped onto s by π. If α consists of 0 steps then s is a start state. Since graph $\mathbf{G}$ is connected, it has a spanning tree T with root v_0. Let u be the state of $DSum^{hp}$ defined by $\pi(u) = s$, $u.tree[\mathsf{v}_0] = \mathsf{e}_0$ and, for $v \neq \mathsf{v}_0$, $u.tree[v]$ equals the unique link of T with target v. Then u is a start state of $DSum^{hp}$. For the induction step, suppose that α ends with a step $s' \xrightarrow{a} s$. By induction hypothesis, there exists a reachable state u' with $\pi(u') = s'$. Let u be the state of $DSum^{hp}$ with $\pi(u) = s$ and $u.tree = u'.tree$. If $u' \xrightarrow{a} u$ is a step then u is reachable and we are done. So assume that $u' \xrightarrow{a} u$ is not a step. Then $a = MSG(e, m)$, for some e and m, and, if we let $v = \mathsf{target}(e)$, $u' \models \neg busy[v] \wedge e \neq tree[v]$. Let t and t' be the states of $DSum^{hp}$ that are identical to u and u', respectively, except that $t.tree[v] = t'.tree[v] = e$. Then $t' \xrightarrow{a} t$ and $\pi(t) = s$. Thus, in order to prove the induction step it suffices to show that t' is reachable. By Lemma 5, $u' \models T_1(w)$, for all w. Therefore, by definition of t', $t' \models T_1(w)$, for all $w \neq v$. Since $s' \models \neg busy[v]$ and s' is reachable and enables a MSG step, $s' \models Init(v)$ by $C_3(e)$. This implies $t' \models Init(v)$, and therefore $t' \models T_1(v)$. We prove that t' satisfies the three conjuncts of T_2:

1. First we prove $v \neq \mathsf{v}_0$ by contradiction. Assume $v = \mathsf{v}_0$. Then, using $s \models Init(v)$ and $s \models I_5$, we conclude $e = \mathsf{e}_0$. By Lemma 6, $u' \models tree[\mathsf{v}_0] = \mathsf{e}_0$. This contradicts the fact $u' \models e \neq tree[v]$. Since $v \neq \mathsf{v}_0$ and $u' \models tree[\mathsf{v}_0] = \mathsf{e}_0$, $t' \models tree[\mathsf{v}_0] = \mathsf{e}_0$.

2. By construction, $t' \models v = \mathsf{target}(tree[v])$. For $w \neq v$, $t' \models w = \mathsf{target}(tree[w])$ follows from the fact that, by Lemma 6, $u' \models w = \mathsf{target}(tree[w])$.

3. If we consider the graph with nodes $\mathbf{V}$ and links $\{t'.tree[v] \mid v \in \mathbf{V}/\{\mathsf{v}_0\}\}$, then clearly each node has one incoming link, except v_0, which has no incoming link. Therefore in order to prove that this graph is a tree, it suffices to show that it has no cycles. We know that the graph with nodes $\mathbf{V}$ and links $\{u'.tree[v] \mid v \in \mathbf{V}/\{\mathsf{v}_0\}\}$ is a tree and therefore has no cycles. Since the only difference between the two graphs is the incoming edge of v, any cycle of $\{t'.tree[v] \mid v \in \mathbf{V}/\{\mathsf{v}_0\}\}$ contains v. But such a cycle cannot exist, since $t' \models Init(v)$ and $t' \models \neg Init(w)$ for all nodes w from which v can be reached by a nonempty path. The proof of this last fact is by induction on the length of the path. For the induction base, note that $t' \models mq[e] \neq \mathsf{empty}$ since t' enables a $MSG(e, m)$ step. Further $v \neq \mathsf{v}_0$ by (1), and thus $e \neq \mathsf{e}_0$. Now use $t' \models C_4(e)$ to conclude $t' \models \neg Init(\mathsf{source}(tree[v]))$. For the induction step, let w be a node with $t' \models \neg Init(w)$. Then $t' \models par[w] = tree[w]$ since $t' \models T_1(w)$. Hence $t' \models \neg Init(\mathsf{source}(tree[w]))$ since $t' \models C_5(w)$.

Now use Lemma 8 to conclude that t' is reachable. □

3.4 A Refinement

In this subsection we will prove the existence of a refinement from $DSum^{hp}$ to S. For this we need two final invariants, which state that non-unit messages can only travel on the reversed spanning tree, and that there is a conservation of weight in the network.

Lemma 10 *For all reachable states of $DSum^{hp}$ and for all e,*

$$\mathsf{head}(mq[e]) \neq 0 \quad \rightarrow \quad e = tree[\mathsf{source}(e)]^{-1}$$

Lemma 11 *For all reachable states of $DSum^{hp}$:*

$$\begin{array}{rcl}\sum_{v\in\mathbf{V}} \mathsf{weight}(v) & = & \sum_{\{v\in\mathbf{V}\mid Init(v)\}} \mathsf{weight}(v)\\ & & + \sum_{\{v\in\mathbf{V}\mid busy[v]\}} total[v]\\ & & + \sum_{\{e\in\mathbf{E}\mid mq[e]\neq\mathsf{empty}\}} \mathsf{head}(mq[e])\\ & & + \text{ if } Done(\mathsf{v}_0) \text{ then } total[\mathsf{v}_0] \text{ else } 0\end{array}$$

Theorem 12 *The function r from states of $DSum^{hp}$ to states of S defined by*

$$r(s) \models done \quad\Leftrightarrow\quad s \models Done(\mathsf{v}_0)$$

is a refinement from $DSum^{hp}$ to S.

Proof: For any start state s of $DSum^{hp}$, $s \models \neg Done(\mathsf{v}_0)$, and for the unique start state u of S, $u \models \neg done$. Hence r satisfies condition (a) in the definition of a refinement.

To prove condition (b), observe that for all reachable states s and for all v,

$$D(v) \quad \triangleq \quad v \neq \mathsf{v}_0 \wedge rcvd(\mathsf{v}_0) = \mathsf{size}(\mathsf{to}(\mathsf{v}_0)) \;\rightarrow\; Done(v)$$

Because suppose that $v \neq \mathsf{v}_0$ and $s \models rcvd(\mathsf{v}_0) = \mathsf{size}(\mathsf{to}(\mathsf{v}_0))$. By induction on the length of the path from v_0 to v in $s.tree$ we prove that $s \models Done(v)$. For the induction base, suppose that $s \models \mathsf{source}(tree[v]) = \mathsf{v}_0$. Then, by $C_2(\mathsf{v}_0)$, $s \models rcvd[tree[v]^{-1}] = 1$. By $I_6(v)$, this implies $s \models \neg Init(v)$. By $T_1(v)$, this in turn implies $s \models par[v] = tree[v]$. Now $s \models Done(v)$ follows by combination of the derived properties with C_7. The induction step is similar.

For condition (b), suppose $s \xrightarrow{a} s'$ is a step of $DSum^{hp}$ and s is reachable. We distinguish between two cases:

1. a is a MSG or $REPORT$ action. Using invariant D it is easy to prove that $s \models \neg Done(\mathsf{v}_0)$ and $s' \models \neg Done(\mathsf{v}_0)$. Hence $r(s) = r(s')$.

2. $a = RESULT(m)$, for some m. Then $s \models busy[\mathsf{v}_0] \wedge cnt[\mathsf{v}_0] = 0$, so by I_{11}, $Rcvd(\mathsf{v}_0) = \mathsf{size}(\mathsf{to}(v))$. By D, this means that $s \models Done(v)$ for all $v \neq \mathsf{v}_0$. If we combine this fact with C_1, we get $mq[e] = \mathsf{empty}$ for all e. Now Lemma 11 gives $s \models total[\mathsf{v}_0] = \sum_{v\in\mathbf{V}} \mathsf{weight}(v)$. Thus, by the precondition of a, $m = \sum_{v\in\mathbf{V}} \mathsf{weight}(v)$. Clearly $s' \models Done(\mathsf{v}_0)$ and so we can conclude $r(s) \xrightarrow{a} r(s')$.

□

3.5 Inclusion of Fair Traces

The fact that $traces(DSum) \subseteq traces(S)$ does not imply $fairtraces(DSum) \subseteq fairtraces(S)$. It might be that $DSum$ does not produce any output but instead performs an infinite sequence of consecutive internal actions or gets into a state of deadlock before an output step has been done. However, using Lemma 4, we can prove the absence of divergent computation:

Lemma 13 *I/O automaton $DSum^h$ has no infinite executions.*

Proof: Define the state function *Norm* as follows:

$$Norm \triangleq \sum_{e \in \mathbf{E}} 2.rcvd[e] + \mathsf{len}(mq[e])$$

Since both sending and receiving a value increases *Norm*, each step of $DSum^h$ with label *MSG* or *REPORT* increases *Norm*. By C_1, *Norm* can be at most $2 \cdot \mathsf{size}(\mathbf{E})$, for any reachable state. Therefore there can be at most finitely many steps labeled by an internal actions in any execution of $DSum^h$. Since *RESULT* steps change the value of $busy[\mathsf{v}_0]$ from true to false, there can be at most one such step after the last internal step. □

The proof that $DSum^h$ has no premature deadlocks is slightly more involved.

Lemma 14 *If a reachable state of* $DSum^h$ *has no outgoing steps then* $Done(\mathsf{v}_0)$ *holds in that state.*

Proof: (Sketch) Suppose that some given state is deadlocked. Then no message can be in transit on the spanning tree or in e_0, otherwise a *MSG* step would be enabled. This implies, by C_6 and I_5, that $\neg Init(v)$ for all nodes v. This in turn implies that no message can be in transit on *any* link it the network (otherwise a *MSG* action would be enabled). Next we use I_7 to infer that exactly one message has been sent on each link in the network, except those on the reversed spanning tree. Finally, we prove for all nodes v of the network, starting with the leaves of the tree, that v has received a message over all incoming links; since no *REPORT* or *RESULT* action is enabled in v this implies $Done(v)$. □

Theorem 15 $fairtraces(DSum) \subseteq fairtraces(S)$.

Proof: (Sketch) The existence of a strong history relation from *DSum* to $DSum^h$ together with Lemmas 13 and 14 guarantee that *DSum* has no infinite executions, or maximal executions consisting of internal actions only. Combined with $traces(DSum) \subseteq traces(S)$ this implies the theorem. □

4. Concluding Remarks

History relations together with refinements form a complete proof method for trace inclusion if the abstract automaton is deterministic [14]. Since I/O automaton S is trivially deterministic, this means that at least in theory there is no need to use prophecy variables in the correctness proof of *DSum*. In fact, it is not so difficult to eliminate the prophecy variable construction from this paper. The key step is to establish as an additional invariant that for all reachable states of $DSum^h$ the set $\{par[v] \mid v \neq \mathsf{v}_0 \wedge \neg Init(v)\}$ forms a tree with root v_0. This alternative proof is even slightly shorter than the proof outlined in this paper. However, I do not think that this is an argument against the use of the prophecy variable *tree*. This auxiliary variable formalizes an important intuition about the algorithm, namely that in each execution a spanning tree is constructed. By fixing this tree, the prophecy variable makes it conceptually simpler to reason about the algorithm.

Since forward simulations form a complete proof method for trace inclusion if the abstract automaton is deterministic [14], the history variable *rcvd* can be eliminated from the proof of this paper in favor of a forward simulation relation. But again, even though this will probably lead to a small reduction in the size of the proof, there are good reasons to keep this auxiliary variable. In the intuitive reasoning about the protocol the number of messages received over the links plays an important role, and the history variable construction makes it possible to formalize this reasoning.

The verification of this paper has not yet been proof-checked by computer. I think that it will be worthwhile to do this, building on earlier work of [20, 6, 17, 3]. An interesting question here is whether the correctness of the history variable construction can be verified fully automatically by a theorem prover, by simply checking the (trivial) proof obligations of a history relation (This would eliminate the need to formalize the meta-theory of history variables.). Another question is whether the prophecy variable construction can be formalized easily, or whether it is simpler to formalize a proof that does not use this construction.

Although I have carried out the verification using the I/O automaton model, it is probably trivial to translate this story to other state based models, such as Lamport's Temporal Logic of Actions [10]. Since liveness issues do not play a role, also a process algebraic verification in a calculus such as μCRL [5] should not be too difficult.

ACKNOWLEDGEMENT
Ching-Tsun Chou outlined how the prophecy variable construction in my proof can be avoided and suggested many improvements. Thanks to David Griffioen for proofreading.

REFERENCES

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison-Wesley, 1988.
3. C.-T. Chou. Mechanical verification of distributed algorithms in higher-order logic. In T.F. Melham and J. Camilleri, editors, *Proceedings 7^{th} International Workshop on Higher-Order Logic and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 158–176. Springer-Verlag, 1994. A revised version will appear in *The Computer Journal*, 1995.
4. C.-T. Chou. Practical use of the notions of events and causality in reasoning about distributed algorithms. CS Report #940035, UCLA, October 1994. Available via anonymous ftp at the URL ftp://ftp.cs.ucla.edu/pub/chou/nil.ps.
5. J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computer Science, pages 231–250. Springer-Verlag, 1993.
6. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings International Workshop TYPES'93,* Nijmegen, The Netherlands, May 1993, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer-Verlag, 1994. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.
7. B. Jonsson. Compositional specification and verification of distributed systems. *ACM*

Transactions on Programming Languages and Systems, 16(2):259–303, March 1994.

8. S.S. Lam and A.U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.
9. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.
10. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, March 1994.
11. P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna, June 1968.
12. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
13. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
14. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part I: Untimed systems. Report CS-R9313, CWI, Amsterdam, March 1993. Also, MIT/LCS/TM-486.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. To appear in *Information and Computation.*
15. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
16. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
17. T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proceedings International Workshop TYPES'94*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.
18. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
19. A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(2):23–35, January 1983.
20. J. Søgaard-Andersen, S. Garland, J. Guttag, N.A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification,* Elounda, Greece, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

A. I/O AUTOMATA AND SIMULATIONS

In this appendix we give a brief account of those parts of I/O automata theory that we need for the purposes of the paper. For a more extensive introduction to the I/O automata model we refer to [12, 13].

A.1 I/O automata

An *action signature* S is a triple $(in(S), out(S), int(S))$ of three disjoint sets of respectively *input actions*, *output actions* and *internal actions.* The derived sets of *external actions*, *locally controlled actions* and *actions* of S are defined respectively by

$$\begin{aligned} ext(S) &= in(S) \cup out(S), \\ local(S) &= out(S) \cup int(S), \\ acts(S) &= in(S) \cup out(S) \cup int(S). \end{aligned}$$

An *I/O automaton* A (or *input/output automaton*) consists of the following five components:

- an action signature $sig(A)$
  (we will write $in(A)$ for $in(sig(A))$, $out(A)$ for $out(sig(A))$, etc.),
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of *transitions*, with the property that for every state s and input action a in $in(A)$ there is a transition (s, a, s') in $steps(A)$,
- a partition $part(A)$ of $local(A)$ in at most countably many equivalence classes.

We let s, s', u, u',.. range over states, and a,.. over actions. We write $s \xrightarrow{a}_A s'$, or just $s \xrightarrow{a} s'$ if A is clear from the context, as a shorthand for $(s, a, s') \in steps(A)$.

An action a is said to be *enabled* in a state s, if $s \xrightarrow{a} s'$ for some s'. Since every input action is enabled in every state, I/O automata are said to be *input enabled.* The intuition behind the input-enabling condition is that input actions are under control of the environment, and that the system that is modeled by an I/O automaton cannot prevent the environment from doing these actions. The partition $part(A)$ describes, what intuitively are the 'components' of the system, and will be used to define fairness.

A.2 Traces and fair traces

Let A be an I/O automaton. An *execution fragment* of A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of A, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment that begins with a start state. A state s of A is *reachable* if it is the final state of some finite execution of A.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of A. Then $trace(\alpha)$, the *trace* of α, is the subsequence of $a_1 a_2 \cdots$ consisting of the external actions of A. With $traces(A)$ we denote the set of traces of executions of A. For s, s' states of A and β a finite sequence of external actions of A, we define $s \overset{\beta}{\Longrightarrow}_A s'$ iff A has a finite execution fragment with first state s, last state s' and trace β.

A *fair execution* of an I/O automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part(A)$:

1. If α is finite, then no action of C is enabled in the final state of α.

2. If α is infinite, then either α contains infinitely many occurrences of actions from C, or α contains infinitely many occurrences of states in which no action from C is enabled.

A fair execution gives fair turns to each class of $part(A)$, and therefore to each component of the system being modeled. A state of A is said to be *quiescent* if only input actions are enabled in this state. Intuitively, in a quiescent state the system is waiting for an input from the environment. A finite execution is fair if and only if its final state is quiescent. We denote the set of traces of fair executions of A by $fairtraces(A)$.

A.3 Simulations

Below we review some basic definitions and results concerning simulation proof techniques. For a more extensive introduction we refer to [14].

Let A and B be I/O automata with the same input and output actions, respectively.

1. A *refinement* from A to B is a function r from states of A to states of B that satisfies the following two conditions:
   (a) If s is a start state of A then $r(s)$ is a start state of B.
   (b) If $s \xrightarrow{a}_A s'$ and both s and $r(s)$ are reachable, then $r(s) \overset{\beta}{\Longrightarrow}_B r(s')$, where $\beta = trace((s, a, s'))$.
2. A *forward simulation* from A to B is a relation between states of A and states of B that satisfies the following two conditions:
   (a) If s is a start state of A then there exists a start state u of B with $(s, u) \in f$.
   (b) If $s \xrightarrow{a}_A s'$, $(s, u) \in f$ and s and u are reachable, then there exists a state u' of B such that $u \overset{\beta}{\Longrightarrow}_B u'$ and $(s', u') \in f$, where $\beta = trace((s, a, s'))$.
3. A *history relation* from A to B is a forward simulation from A to B whose inverse is a refinement from B to A.
4. A *backward simulation* from A to B is a relation between states of A and states of B that satisfies the following three conditions:
   (a) If s is a start state of A and u is a reachable state of B with $(s, u) \in b$, then u is a start state of B.
   (b) If $s \xrightarrow{a}_A s'$, $(s', u') \in b$ and s and u' are reachable, then there exists a reachable state u of B such that $u \overset{\beta}{\Longrightarrow}_B u'$ and $(s, u) \in b$, where $\beta = trace((s, a, s'))$.
   (c) If s is a reachable state of A then there exists a reachable state u of B with $(s, u) \in b$.
5. A *prophecy relation* from A to B is a backward simulation from A to B whose inverse is a refinement from B to A.

A refinement, forward simulation, etc. is called *strong* if in each case where one automaton is required to simulate a step from the other automaton, this is possible with an execution fragment consisting of *exactly* one step.[4] A relation R over S_1 and S_2 is *image-finite* if for all elements s_1 of S_1 there are only finitely many elements s_2 of S_2 such that $(s_1, s_2) \in R$.

Theorem 16 ([14]) *Let A and B be I/O automata with the same input and output actions, respectively.*

1. *If there is a refinement from A to B then $traces(A) \subseteq traces(B)$.*
2. *If there is a forward simulation from A to B then $traces(A) \subseteq traces(B)$.*
3. *If there is a history relation from A to B then $traces(A) = traces(B)$.*
4. *If there is an image-finite backward simulation from A to B then $traces(A) \subseteq traces(B)$.*
5. *If there is an image-finite prophecy relation from A to B then $traces(A) = traces(B)$.*

[4]Here we use the word "strong" in the sense of [16]. Actually, the notions of simulation that we consider here are "weak" in the sense of [14] since their definitions include reachability conditions.