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

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 (in this context we can take "small" to mean "fits on one or two pages").

Not everybody agrees with this view, however, and frequently one can hear claims that existing approaches cannot deal (or 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 [3], 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.

To illustrate the shortcoming of the assertional methods of [2, 5, 6, 7, 8, 10, 13], Chou discusses a variant of Segall's PIF (Propagation of Information with Feedback) protocol [18]. A complex and messy classical proof of this algorithm is contrasted with a slightly simpler but definitely more structured proof based on the new method advocated by the author.

I think that Chou's view of existing assertional methods is much too pessimistic. First of all these methods are not ad-hoc, but 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 prove 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 relation (and thereby the safety properties) has been established, this relation 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 "classical" assertional methods, I present in this paper a verification of the algorithm discussed by Chou [3]. Altogether, it took me about two hours to come up with a detailed sketch of the proof (during a train ride from Leiden to Eindhoven), and less than two weeks to work it out and write this paper. The proof is completely routine, except for a few nice invariants and the idea to use a prophecy variable. Unlike history variables, which date back to the sixties [9], 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, such as [17], new methods will pay off and lead to shorter proofs that are closer to intuition. This paper shows that, unlike what is claimed by Chou [3], the old methods still work very well for a variant of Segall's PIF protocol.

2 Labeled Transition Systems and Simulations

In this paper we use a very simple and well-known transition system model. The model is a simplified version of the I/O automata model [10, 11]: it does not deal with fairness or other forms of liveness and there is no distinction between input and output actions. In this section we review some basic definitions and results concerning automata and simulation proof techniques. For a more extensive introduction we refer to [12].

DEFINITION 1 A *labeled transition system* or *automaton* A consists of four components:

- A (finite or infinite) set $states(A)$ of states.
- A nonempty set $start(A) \subseteq states(A)$ of start states.
- A pair $(ext(A), int(A))$ of disjoint sets of external and internal actions, respectively. The derived set $acts(A)$ of actions is defined as the union of $ext(A)$ and $int(A)$.
- A set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of steps.

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 *execution fragment* of an automaton A is a finite or infinite alternating sequence, $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$, 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, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. The function *first* gives the first state of an execution fragment and, for finite execution fragments, the function *last* gives the final state. An *execution* of A is an execution fragment that begins with a start state. A state s of A is *reachable* if $s = last(\alpha)$ for some finite execution α of A.

The *trace* of an execution fragment α, written $trace(\alpha)$, is the sequence of external actions occurring in α. A sequence β of actions is a *trace* of automaton A if there is an execution α of A with $\beta = trace(\alpha)$. The set of traces of A is denoted by $traces(A)$. Suppose s and s' are states of A, and β is a finite sequence of external actions of A. We write $s \overset{\beta}{\Longrightarrow}_A s'$, or just $s' \overset{\beta}{\Longrightarrow} s$, if A has a finite execution fragment α with $first(\alpha) = s$, $trace(\alpha) = \beta$ and $last(\alpha) = s'$.

DEFINITION 2 Let A and B be automata with the same external actions.

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.[1]

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 1 *Let A and B be automata with the same external actions.*

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 $traces(A) = traces(B)$.*

3 DESCRIPTION OF THE ALGORITHM

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, i.e., $(v,w) \in \mathbf{E} \Leftrightarrow (w,v) \in \mathbf{E}$, and also strongly 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 unit and that there is a binary operator $\circ$

[1] Here we use the word "strong" in the sense of [14]. Actually, the notions of simulation that we consider here are *weak* in the sense of [12] since their definitions include reachability conditions.

on $\mathbf{M}$, such that $(\mathbf{M}, \circ, \mathsf{unit})$ is an Abelian monoid (so $\circ$ is commutative and associative and has unit element unit).

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 rephrasing of an algorithm described by Chou [3], which in turn is a variant of Segall's PIF (Propagation of Information with Feedback) protocol [18].

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.[2] When an arbitrary node v receives a first message, it marks the node w from which this message was received. It then sends a unit 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 automaton *SUM* using the standard precondition/effect style of the I/O automata model [10, 11, 4]. 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 this 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 together with 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 $\mathsf{target}(e)$, 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 *SUM* are interpretations of five state variables in their domains. The first four of these variables represent the values of program variables at each node:

[2]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
$REPORT$
External: $RESULT$

State Variables: $busy \in \mathbf{V} \to \mathbf{Bool}$
$parent \in \mathbf{V} \to \mathbf{E}$
$total \in \mathbf{V} \to \mathbf{M}$
$cnt \in \mathbf{V} \to \mathbf{Int}$
$mq \in \mathbf{E} \to \mathbf{M}^*$

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

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

$REPORT(e : \mathbf{E}, m : \mathbf{M})$
Precondition:
$v = \mathsf{source}(e) \neq \mathsf{v}_0 \wedge busy[v] \wedge cnt[v] = 0 \wedge e^{-1} = parent[v] \wedge m = total[v]$
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. Automaton SUM.

- *busy* tells for each node whether or not it is currently participating in the protocol; initially $busy[v]$ equals **false** for each v;
- *parent* 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.

State variable mq, finally, represents the contents of the message queue for each link. Initially, $mq[e]$ is **empty** for each link e except e_0.

Automaton *SUM* has three types of actions: an action *MSG*, which describes the receipt and processing of a message, an action *REPORT*, by which a non root node sends the final value that it has computed to its parent, and an action *RESULT*, which is the last action of the algorithm, used by the root node to output the final result of the computation.

4 Correctness Proof

The correctness property Φ of *SUM* that we want to establish is that each maximal execution of the automaton consists of a finite number of internal actions followed by the single output action $RESULT(\sum_{v \in \mathbf{V}} \mathsf{weight}(v))$.

Intuitively, propagation of messages occurs in two phases. First **unit** messages are sent from node v_0 into the network, and then partial sums flow back from the network to v_0. In the first phase a spanning tree is constructed with root v_0 and this spanning tree is used to accumulate values in the second phase.

4.1 Adding a History Variable

A first important observation about the algorithm is that in each run at most one message travels on each link. In order to state this property formally as an invariant, we add a so-called "history variable" *sent* to automaton *SUM* that records for each link e how many messages have been sent on e. Figure 2 describes the automaton SUM^h obtained in this way. Variable *sent* is an auxiliary/history variable in the sense of Owicki and Gries [16] because it does not occur in conditions nor at the right-hand-side of assignments to other variables. Clearly, adding *sent* does not change the behavior of automaton *SUM*. This can be formalized via the following trivial lemma, which in turn implies that *SUM* satisfies correctness property Φ if and only if SUM^h does.

Lemma 2 *The inverse of the projection function that maps states from SUM^h to states of SUM is a strong history relation from SUM to SUM^h.*

Invariant 1 below gives a basic sanity property of SUM^h: at any time the number of messages in a link is at most equal to the number of messages that have been sent on that link.

Internal: MSG
$REPORT$
External: $RESULT$

State Variables: $busy \in \mathbf{V} \to \mathbf{Bool}$
$parent \in \mathbf{V} \to \mathbf{E}$
$total \in \mathbf{V} \to \mathbf{M}$
$cnt \in \mathbf{V} \to \mathbf{Int}$
$mq \in \mathbf{E} \to \mathbf{M}^*$
$sent \in \mathbf{E} \to \mathbf{Int}$

Init: $\bigwedge_v \neg busy[v]$
$\bigwedge_e mq[e] = \text{if } e{=}\mathsf{e}_0 \text{ then append(unit, empty) else empty}$
$\bigwedge_e sent[e] = \text{if } e{=}\mathsf{e}_0 \text{ then 1 else 0}$

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

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

$RESULT(m : \mathbf{M})$
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 2. Automaton SUM^h obtained from SUM by adding history variable $sent$.

INVARIANT 1 *For all reachable states of* SUM^h *and for all* e*:*

$$\mathsf{len}(mq[e]) \leq sent[e]$$

At first sight, Invariant 2 below may look a bit complicated. It is however easy to give intuition for it. The key part of the invariant is the first conjunct, which states that at most one message travels on each link. The other conjuncts are only needed to get the induction to work in the invariant proof. The second and third conjunct imply that if in a *MSG* step a value is sent into some channel, this channels must have been empty in the start state of that step. The fourth conjunct allows to prove a similar property for *REPORT* steps. The routine proof of Invariant 2, which has been omitted here, uses Invariant 1.

INVARIANT 2 *For all reachable states of* SUM^h *and for all* v *and* e*:*

$\wedge\ sent[e] \leq 1$
$\wedge\ \mathsf{len}(mq[\mathsf{e}_0])=1 \rightarrow (\forall f \in \mathsf{from}(\mathsf{v}_0)/\{\mathsf{e}_0\} : sent[f]=0)$
$\wedge\ v \neq \mathsf{v}_0 \wedge \neg busy[v] \wedge e \in \mathsf{to}(v) \wedge \mathsf{len}(mq[e])=1 \rightarrow (\forall f \in \mathsf{from}(v) : sent[f]=0)$
$\wedge\ v \neq \mathsf{v}_0 \wedge busy[v] \wedge e = parent[v] \rightarrow sent[e^{-1}]=0$

Invariant 2 is quite powerful and implies in particular that the algorithm will always terminate.

COROLLARY 3 *Automaton* SUM^h *has no infinite executions.*

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

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

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

A next property that we will established is that each node can be activated only once in any run of the algorithm. We say that node v is *activated* in a step if $busy[v]$ changes from false to true in that step. This implies that v has been activated iff it has received at least one message. The number of messages received by a node v equals the number of messages that have been sent to v minus the number of messages still in transit, and is therefore given by the state function:

$$Received(v) \triangleq \sum_{e \in \mathsf{to}(v)} sent[e] - \mathsf{len}(mq[e])$$

The following Invariant 3 gives a characterization of the value of $Received(v)$ for reachable states. The proof is straightforward and uses Invariant 2.

INVARIANT 3 *For all reachable states of* SUM^h *and for all* v*:*

$$\begin{array}{ll} \wedge\ busy[v] & \rightarrow\ Received(v) = \mathsf{size}(\mathsf{to}(v)) - cnt[v] > 0 \\ \wedge\ \neg busy[v] & \rightarrow\ Received(v) = 0 \vee Received(v) = \mathsf{size}(\mathsf{to}(v)) \end{array}$$

Invariants 2 and 3 together imply that each node is activated at most once in each execution. Because suppose that in some reachable state some node v is both inactive and activated. This means $\neg busy[v] \wedge Received(v) > 0$. Then Invariant 3 gives $Received(v) = \mathsf{size}(\mathsf{to}(v))$. But this implies that no *MSG* action can be enabled, because this would violate Invariant 2.

We conclude this subsection with two simple invariants that we will use later on.

INVARIANT 4 *For all reachable states of* SUM^h *and for all* v*:*

$$Received(v) > 0 \quad \rightarrow \quad v = \mathsf{target}(parent[v])$$

INVARIANT 5 *For all reachable states of* SUM^h *and for all* e*:*

$$e \neq \mathsf{e}_0 \wedge mq[e] \neq \mathsf{empty} \quad \rightarrow \quad Received(\mathsf{source}(e)) > 0$$

4.2 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 *prophecy variables* of Abadi and Lamport [1] allows us to drastically reduce the nondeterminism of the algorithm or, more precisely, to push nondeterminism backwards to the initial state. We add to SUM^h a new variable *tree*, which records an initial guess of the full spanning tree and is used to enforce that the actual tree that is constructed during execution is equal to this initial guess. Figure 3 describes the automaton SUM^{hp} obtained in this way. In Figure 3, tree is the function that tells for each set of links whether or not it is a tree. More formally, for $T \subseteq \mathbf{E}$ and $E = \{\mathsf{source}(e), \mathsf{target}(e) \mid e \in T\}$, $\mathsf{tree}(T) = \mathsf{true}$ iff either $T = \emptyset$ or there exists a node $v \in E$ such that for all $v' \in E$ there is a unique path of links in T leading from v to v'.

In order to show that *tree* is a prophecy variable in the sense of [1, 12], we establish a prophecy relation from SUM^h to SUM^{hp}. For this, we need three more invariants. The proof of Invariant 6 uses Invariants 3, 4 and 5. Invariants 7 and 8 are completely trivial.

Internal: MSG
$REPORT$
External: $RESULT$

State Variables: $busy \in \mathbf{V} \to \mathbf{Bool}$
$parent \in \mathbf{V} \to \mathbf{E}$
$total \in \mathbf{V} \to \mathbf{M}$
$cnt \in \mathbf{V} \to \mathbf{Int}$
$mq \in \mathbf{E} \to \mathbf{M}^*$
$sent \in \mathbf{E} \to \mathbf{Int}$
$tree \in \mathbf{V} \to \mathbf{E}$

Init: $\bigwedge_v \neg busy[v]$
$\bigwedge_e mq[e] = \text{if } e{=}\mathsf{e}_0 \text{ then } \mathsf{append}(\mathsf{unit}, \mathsf{empty}) \text{ else } \mathsf{empty}$
$\bigwedge_e sent[e] = \text{if } e{=}\mathsf{e}_0 \text{ then } 1 \text{ else } 0$
$\bigwedge_v tree[\mathsf{v}_0] = \mathsf{e}_0 \wedge v = \mathsf{target}(tree[v]) \wedge \mathsf{tree}(\{tree[v] \mid v \in \mathbf{V}/\{\mathsf{v}_0\}\})$

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

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

$RESULT(m : \mathbf{M})$
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 3. Automaton SUM^{hp} obtained from SUM^h by adding prophecy variable $tree$.

INVARIANT 6 *Let T be the state function defined by*

$$T \triangleq \{parent[v] \mid v \neq \mathsf{v}_0 \wedge Received(v) > 0\}$$

Then $\mathsf{tree}(T)$ *holds for all reachable states of* SUM^h.

INVARIANT 7 *For all reachable states of* SUM^{hp} *and for all* v*:*

$$Received(v) > 0 \quad \rightarrow \quad parent[v] = tree[v]$$

INVARIANT 8 *For all reachable states of* SUM^{hp} *and for all* v*:*

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

LEMMA 4 *The inverse of the projection function π that maps states of SUM^{hp} to states of SUM^h is a strong image-finite prophecy relation from SUM^h to SUM^{hp}.*

PROOF: Mapping π is trivially a strong refinement from SUM^{hp} to SUM^h. 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 SUM^h and u is a reachable state of SUM^{hp} with $\pi(u) = s$. Then it follows by Invariant 8 that u is a start state of SUM^{hp}.

To prove that π^{-1} satisfies conditions (b) and (c) we need the following claim: a state u of SUM^{hp} is reachable iff $\pi(u)$ is reachable and u satisfies the properties of Invariants 7 and 8. Direction "$\Rightarrow$" of this claim follows by induction on the length of the shortest execution to u, and uses the fact that π is a strong refinement together with Invariants 7 and 8. Direction "$\Leftarrow$" of the claim follows by induction on the length of the shortest execution to $\pi(u)$.

Using the claim, it is routine to prove condition (b). Condition (c) follows from the claim together with Invariant 6. □

Note that as a direct corollary of Lemma 4 all invariants of SUM^h are also invariants of SUM^{hp}.

4.3 A Refinement

In this subsection we will prove that there exists a refinement from automaton SUM^{hp} to the automaton S defined in Figure 4. Automaton S is extremely simple. It has only two states: an initial state where *done*=false and a final state where *done*=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.

Define state functions *Init* and *Done* by

$$\begin{aligned} Init(v) &\triangleq \neg busy[v] \wedge Received(v) = 0 \\ Done(v) &\triangleq \neg busy[v] \wedge Received(v) = \mathsf{size}(\mathsf{to}(v)) \end{aligned}$$

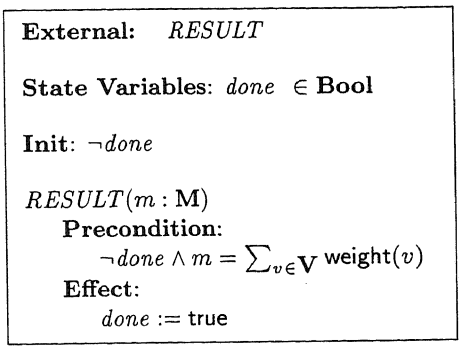

FIGURE 4. Automaton S.

As a consequence of Invariant 3, each reachable state of SUM^{hp} satisfies, for each v, either $Init(v)$ or $busy[v]$ or $Done(v)$. In order to establish a refinement from SUM^{hp} to S, we again need two extra invariants. Invariant 9 states that, until the moment where computation has finished, there is a conservation of weight in the network. Invariant 10 allows us to prove that in a state where $RESULT$ is enabled, $Done(v)$ holds for all nodes except v_0.

INVARIANT 9 *For all reachable states of* SUM^{hp}*:*

$$\neg Done(v_0) \rightarrow \sum_{v \in \mathbf{V}} \mathsf{weight}(v) = \sum_{\{v \in \mathbf{V} \mid Received(v)=0\}} \mathsf{weight}(v) + \sum_{\{v \in \mathbf{V} \mid busy[v]\}} total[v] + \sum_{\{e \in \mathbf{E} \mid mq[e] \neq \mathsf{empty}\}} \mathsf{head}(mq[e])$$

INVARIANT 10 *For all reachable states of* SUM^{hp} *and for all* v *and* e*:*

$$v \neq v_0 \wedge e = tree[v] \wedge sent[e^{-1}] = 1 \rightarrow Done(v)$$

LEMMA 5 *The function* r *from states of* SUM^{hp} *to states of* S *given by*

$$r(s) \models done \Leftrightarrow s \models Done(v_0)$$

is a refinement from SUM^{hp} *to* S*.*

4.4 Absence of Deadlock

The existence of a refinement mapping from SUM^{hp} to S does not guarantee that automaton SUM^{hp} will produce any output: the automaton still may

have an infinite loop of internal actions or get into a state of deadlock before an output step has been done. We can easily prove the absence of infinite loops by using the result of Corollary 3 that SUM^h has no infinite executions and the fact that there is a strong prophecy relation from SUM^h to SUM^{hp}. The proof that SUM^{hp} has no premature deadlocks is more involved and requires three additonal invariants.

INVARIANT 11 *For all reachable states of* SUM^{hp}, $sent[\mathsf{e}_0] = 1$.

INVARIANT 12 *For all reachable states of* SUM^{hp} *and for all* v *and* e*:*

$$e = tree[v] \wedge Init(v) \wedge mq[e] = \mathsf{empty} \quad \rightarrow \quad Init(\mathsf{source}(e))$$

INVARIANT 13 *For all reachable states of* SUM^{hp} *and for all* v *and* e*:*

$$\neg Init(v) \wedge \mathsf{source}(e) = v \wedge e^{-1} \neq tree[v] \quad \rightarrow \quad sent[e] = 1$$

LEMMA 6 *A reachable state of* SUM^{hp} *has no outgoing steps if and only if* $Done(\mathsf{v}_0)$ *holds in that state.*

PROOF: (Sketch)

"$\Leftarrow$" If $Done(\mathsf{v}_0)$ holds then we can prove using Invariant 10 that $Done(v)$ holds for all nodes v. Then Invariants 2 and 3 together imply that no message is in transit. Consequently, no step of SUM^{hp} is enabled.

"$\Rightarrow$" Suppose that some given state is deadlocked. Then no message can be in transit on the spanning tree, otherwise a *MSG* step would be enabled. This implies, by Invariants 11 and 13, that $\neg Init(v)$ holds 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 Invariant 13 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 7 *Automaton SUM satisfies property* Φ.

PROOF: Follows from the fact that SUM^{hp} satisfies Φ and the existence of a strong history relation from *SUM* to SUM^h and a strong prophecy relation from SUM^h to SUM^{hp}. □

5 CONCLUDING REMARKS

The verification of this paper has not yet been proof-checked by computer, but I expect that this will be a routine exercise, building on earlier work on mechanical checking of I/O automata proofs [19, 4, 15]. Although I have carried out the verification using a simple version of 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 [8].

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. Chou. Practical use of the notions of events and causality in reasoning about distributed algorithms. CS Report #940035, UCLA, October 1994.
4. 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.
5. B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.
6. S.S. Lam and A.U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.
7. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.
8. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, March 1994.
9. P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna, June 1968.
10. 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.
11. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
12. 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.*
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
14. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
15. T. Nipkow and K. Slind. I/O automata in Isabelle/HOL, 1994. Draft paper.
16. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
17. P. Humblet R. Gallager and P. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

18. A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(2):23–35, January 1983.
19. 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.