Mechanical Verification of Distributed Algorithms in Higher-Order Logic*

CHING-TSUN CHOU

Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90024, USA
Email: chou@cs.ucla.edu

The only practical way to verify the correctness of distributed algorithms with a high degree of confidence is to construct machine-checked, formal correctness proofs. In this paper we explain how to do so using HOL—an interactive proof assistant for higher-order logic developed by Gordon and others. First, we describe how to build an infrastructure in HOL that supports reasoning about distributed algorithms, including formal theories of predicates, temporal logic, labeled transition systems, simulation of programs, translation of properties, and graphs. Then we demonstrate, via an example, how to use the powerful intuition about events and causality to guide and structure correctness proofs of distributed algorithms. The example used is the verification of PIF (propagation of information with feedback), which is a simple but typical distributed algorithm due to Segall.

1 INTRODUCTION

Distributed algorithms, as exemplified by those in [15, 33], have to operate in the face of total *asynchrony*: there is no finite bound on how widely component speeds may vary both spatially and temporally in a distributed system. The asynchronous interaction of concurrent activities can produce *nondeterministic* behaviors that are too numerous to be adequately tested and too complex for informal reasoning about them to be reliable. The only practical way to verify the correctness of distributed algorithms with a high degree of confidence is to construct machine-checked, formal correctness proofs. The aim of this paper is to explain how to do so.

The proof construction tool we use is HOL, which is an interactive proof assistant for higher-order logic developed by Gordon and others [18]. (A very brief introduction to higher-order logic can be found in Section 2; for a more thorough treatment, see [18].) We choose HOL for several reasons. The first is *expressiveness*: higher-order logic supports natural formalizations of distributed algorithms, the data structures they manipulate, the properties they possess, and the modes of reasoning used to prove those properties. The ability to have natural formalizations is important not only because it reduces the risk of having *wrong* formalizations, but also because it helps to communicate the meanings of proofs to the uninitiated. The second is *security*: HOL reduces every proof to the repeated application of a small set of axioms and inference rules, which is more trustworthy than a large set of *ad hoc* proof procedures. Since our foremost concern is correctness, such a high standard of security is very desirable. The third is *programmability*: customized proof procedures can be easily written to automate common patterns of reasoning without compromising the security of HOL. This also encourages the re-use of proof procedures among a group of users. We have found that these characteristics do make HOL a secure and flexible environment in which to construct correctness proofs of distributed algorithms.

Our approach to formalization is strictly *definitional*, meaning that all our theories are developed from the initial theory of HOL without introducing any axioms other than definitions. The definitional approach has two well-known advantages [4]. The first is *consistency*: definitions never introduce any inconsistencies. Since the initial theory of HOL is consistent [18], all our theories are consistent as well. The second is *eliminability*: definitions can, in principle, be eliminated. So an auxiliary definition not involved in the statement of a main theorem can safely be forgotten as far as that theorem is concerned, even if it is used in the proof of that theorem. To be sure, sticking to the definitional approach is sometimes laborious, since even "obvious theorems" must be honestly proved. But the logical security thus gained is well worth the effort, as too many "obvious theorems" have turned out to be false.

In Section 3, we describe how to build an infrastructure in HOL that supports reasoning about distributed algorithms, including formal theories of predicates, temporal logic, labeled transition systems, simulation of programs, translation of properties, and graphs. Here we mention only two highlights. First, a *Fundamental Theorem* (depicted schematically in (1) below) allows one to deduce the satisfaction by a "concrete" program

*This paper is an expanded and fully revised version of [9].

$\Pi^\flat$ of a "concrete" property $P^\flat$ (written $\Pi^\flat \models P^\flat$) from the satisfaction by an "abstract" program $\Pi^\sharp$ of an "abstract" property $P^\sharp$ (written $\Pi^\sharp \models P^\sharp$), provided that we can establish the *simulation* of $\Pi^\flat$ by $\Pi^\sharp$ via a *joint invariant* J (written $\mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp)$) and the *translation* of $P^\sharp$ into $P^\flat$ via the inverse of J (written $P^\sharp \{\!\!\{J^{-1}\}\!\!\} P^\flat$).

$$\begin{tikzcd} \Pi^\sharp \arrow[r, phantom, "\models"] \arrow[d, phantom, "\Downarrow"] & P^\sharp \arrow[d, "P^\sharp \{\!\!\{J^{-1}\}\!\!\} P^\flat"] \\ \Pi^\flat \arrow[u, bend left, "\mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp)"] \arrow[r, phantom, "\models"] & P^\flat \end{tikzcd} \qquad (1)$$

Second, a set of *Reduction Lemmas* allow one to reduce the proof of the translation relation between two complex properties *of the same form* to the proofs of translation relations between their corresponding constituents. A typical example is the Reduction Lemma for implicational properties (the $\Rrightarrow$ operator will be defined in Section 3.1.2):

$$\begin{aligned} &P_1^\flat \{\!\!\{J\}\!\!\} P_1^\sharp \wedge P_2^\sharp \{\!\!\{J^{-1}\}\!\!\} P_2^\flat \\ &\Rightarrow \quad (P_1^\sharp \Rrightarrow P_2^\sharp) \{\!\!\{J^{-1}\}\!\!\} (P_1^\flat \Rrightarrow P_2^\flat) \end{aligned}$$

When applicable, the Fundamental Theorem (1) together with the Reduction Lemmas allows one to carry out a significant portion of the correctness proof of the concrete program $\Pi^\flat$ (including all temporal reasoning) in terms of the abstraction $\Pi^\sharp$. This is desirable not only because $\Pi^\sharp$ is more abstract and easier to reason about than $\Pi^\flat$, but also because different concrete programs may share the same abstraction $\Pi^\sharp$ whose proof can then be reused.

Our notion of simulation is a generalization of that of Milner's [29]. Our formulation of translation is based on a novel interpretation of Hoare triples [19]. Please see Sections 3.3 and 3.4 for details.

Common patterns of reasoning in program verification are often formulated as special-purpose *programming logics*, such as Hoare logic [19] and temporal logic [32], so that these patterns can be expressed succinctly and made more easily applicable. In this work, we borrow some constructs from Chandy and Misra's UNITY [5] and Lamport's TLA [22] (see Section 3.2 for details). In keeping with the definitional approach, we *embed* these constructs in higher-order logic by formalizing their semantics as definitions and proving their properties as theorems of higher-order logic. This approach allows arbitrary mixing of embedded logics with other mathematical theories and hence is very flexible. Furthermore, there is no need to worry about the completeness of inference rules of an embedded logic, for new rules can always be derived from the formalized semantics whenever needed.

To reason about distributed algorithms, a considerable amount of graph theory is indispensable. But formalizing graph theory is *not* trivial, since there has been little tradition of formalization in graph theory due to the concreteness of graphs. We have formalized the graph theory needed in this work and reported the results elsewhere [8]. The graph-theoretic notions used in this paper is summarized in Section 3.5.

In Section 4, we demonstrate, via an example, how to use the powerful intuition about events and causality to guide and structure correctness proofs of distributed algorithms. By *events* we mean the (names of) occurrences of atomic actions in an execution of a distributed system and by *causality* we mean the essential temporal precedence relation between events that is respected by all possible interleavings of concurrent events in that execution. The computation of a distributed system can be viewed as the (generally nondeterministic) unfolding of a causality relation between events. Normally there are much fewer possible causality relations than possible interleavings. Consequently, the event-and-causality view is often an excellent way to visualize the computation of a distributed system. Thus, given a distributed algorithm Π^D, it is often quite easy to write down an "event algorithm" Π^E whose sole purpose is to generate the events and causality relations that Π^D can generate. By taking Π^D as the concrete program and Π^E as the abstract program, the theory of simulation and translation outlined above can be used to deduce properties of Π^D from those of Π^E, which in turn can be derived by reasoning about events and causality in Π^E. Writing down the joint invariant connecting Π^D and Π^E is not hard; it amounts to expressing the current state of Π^D in terms of that of Π^E.

The example used in Section 4 is the verification of PIF (propagation of information with feedback), which is a simple but typical distributed algorithm due to Segall [33]. PIF performs the task of propagating a piece of information from a distinguished node (called the root) to all other nodes in a (connected) computer network and notifying the root after that information has indeed reached every node in the network. PIF also builds, as a by-product, a spanning tree of the network. Though simple, PIF is highly nondeterministic: for *any* spanning tree of the network, there is an execution in which that tree is chosen by PIF. We will show in Section 4.2 that, despite the nondeterminism, the notions of events and causality afford an intuitive way to understand and reason about PIF.

Section 5 surveys related work. Section 6 discusses possible extensions of this work.

2 HIGHER-ORDER LOGIC

Higher-order logic, also known as the simple theory of types [11], generalizes first-order logic by allowing quantification over functions. For example, the induction principle for natural numbers can be expressed in higher-order logic by a single formula:

$$\begin{aligned} &\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n{+}1)) \\ &\qquad \Rightarrow (\forall n.\ P(n)) \end{aligned} \qquad (2)$$

Every term t in higher-order logic has a type τ, written $t : \tau$. Intuitively, t denotes a value belonging to the set of values denoted by τ. For example, in (2), we have:

$$\begin{array}{lcl} n & : & num \\ + & : & num \rightarrow num \rightarrow num \\ P & : & num \rightarrow bool \\ \wedge & : & bool \rightarrow bool \rightarrow bool \end{array}$$

where *num* is the type of natural numbers, *bool* is the type of truth values T and F, and $\alpha \rightarrow \beta$ is the type of functions from α to β, for any types α and β. Note that a formula is just a term of type *bool*. A function $f : \alpha \rightarrow \beta$ can be applied to an argument $a : \alpha'$ only when α and α' are identical; the result $f(a)$ is of type β. For any variable $v : \alpha$ and any term $t : \beta$, the *lambda abstraction* $(\lambda v\,.\,t)$ denotes a function of type $\alpha \rightarrow \beta$ which, when applied to an argument $a : \alpha$, returns $t[a/v]$, the result of substituting a for v in t.

Primitive terms of higher-order logic are of two kinds: *constants* and *variables*. A constant denotes a fixed value of the appropriate type, while a variable denotes an arbitrary value and hence can be bound by a quantifier. For example, in (2), 0, +, $\Rightarrow$ are constants, and n and P are (bound) variables. (There is no free variable in (2).) New constants can be introduced using *definitions*. For example, the definition:

$$\mathsf{sos}(x)(y) \;\triangleq\; x * x + y * y$$

defines a constant sos : $num \rightarrow num \rightarrow num$ that computes the sum of squares of its arguments.

New types of *literals* can be defined using Melham's recursive type definition package [27] (though all literal types used in this paper are *non*-recursive). For example, Melham's package can take a "BNF" of the form:

$$ttt \;::=\; \mathtt{aaa} \mid \mathtt{bbb}(bool) \mid \mathtt{ccc}(num)(num)$$

and define a new type ttt with three constructors:

$$\begin{array}{lcl} \mathtt{aaa} & : & ttt \\ \mathtt{bbb} & : & bool \rightarrow ttt \\ \mathtt{ccc} & : & num \rightarrow num \rightarrow ttt \end{array}$$

such that every value of type ttt equals exactly one of aaa, bbb(x), and ccc$(y)(z)$, for (when applicable) some unique x, y, and z.

The typographic convention used in this section will be generally followed in the sequel: arbitrary types are denoted by lowercase Greek letters, specific types by *slanted* identifiers, variables by *italic* identifiers or capital Greek letters, constants by sans serif identifiers or non-alphabetic symbols, and literals by `typewriter` identifiers. Free variables in a definition or theorem are implicitly universally quantified. The word "iff" means "if and only if".

3 THE INFRASTRUCTURE

3.1 Predicates

A *predicate* P is a function whose type is of the form $\alpha \rightarrow bool$, where α is called the *domain* of P. In this work, predicates are used extensively and in two ways: as *sets* and as *propositions*.

3.1.1 *Predicates as Sets*

A predicate $P : \alpha \rightarrow bool$ can be viewed as the set of values of type α that satisfies P:

$$P \;=\; \{\, x : \alpha \mid P(x) \,\}$$

In other words, we identify a set with its characteristic predicate. With this identification, the usual operations and notations of sets can be applied to predicates as well, such as $\subseteq$, $\cap$, $\cup$, $\backslash$ (set difference), $|P|$ (cardinality), $\{1,2,3\}$, and $\{n \mid n > 3\}$.[1] Also useful are *restricted quantifications* over sets:

$$\begin{array}{lcl} (\forall x :: P\,.\,Q[x]\,) & \triangleq & (\forall x : \alpha\,.\,P(x) \Rightarrow Q[x]\,) \\ (\exists x :: P\,.\,Q[x]\,) & \triangleq & (\exists x : \alpha\,.\,P(x) \wedge Q[x]\,) \end{array}$$

where the $Q[x]$ notation indicates that x may (but does not necessarily!) occur free in Q.

3.1.2 *Predicates as Propositions*

Our method for embedding special-purpose logics, such as temporal logic, in higher-order logic is to use predicates over suitable domains to represent propositions of the embedded logic. Boolean connectives, quantifiers, and validity can all be *lifted* to operate on predicates:

$$\begin{array}{rcl} (\neg\!\!\neg P)(x) & \triangleq & \neg P(x) \\ (P \mathbin{\wedge\!\!\!\wedge} Q)(x) & \triangleq & P(x) \wedge Q(x) \\ (P \mathbin{\vee\!\!\!\vee} Q)(x) & \triangleq & P(x) \vee Q(x) \\ (P \Rrightarrow Q)(x) & \triangleq & P(x) \Rightarrow Q(x) \\ (\forall\!\!\!\forall i\,.\,R[i])(x) & \triangleq & \forall i\,.\,(R[i])(x) \\ (\exists\!\!\!\exists i\,.\,R[i])(x) & \triangleq & \exists i\,.\,(R[i])(x) \\ \Vdash P & \triangleq & \forall x\,.\,P(x) \end{array}$$

Note our convention of using the "doubled" version of a symbol for its lifted counterpart. These lifted operators provide the logical infrastructure for an embedded logic. For example, the following theorem is the lifted version of *modus ponens*:

$$\forall P\,Q\,.\;\Vdash(P \Rrightarrow Q) \;\Rightarrow\; (\Vdash P \;\Rightarrow\; \Vdash Q)$$

[1]We should point out that the "sets" in higher-order logic behave differently from the sets in set theory. A "set" in higher-order logic can contain elements from only a single type, so (for instance) neither $\{7, \mathtt{aaa}\}$ nor $\{5\} \cap \{\mathsf{F}\}$ makes sense in higher-order logic. It also follows that there is a distinct "empty set" for each type in higher-order logic.

Other inference rules of HOL can be similarly lifted to constitute a *lifted logic*. See [6] for how to *uniformly* lift proof procedures of HOL to handle proofs in the lifted logic, based on a *sequent* formulation of lifted validity.

3.2 Programs and Properties

In this work, programs are represented by (fair) *labeled transition systems* and their properties are expressed in an embedded *temporal logic*.

Let σ be a type of *states* and α be a type of *actions*. A labeled transition system is a pair of the form:

$$(\, I : \sigma \rightarrow bool\, ,\ N : \sigma \times \alpha \times \sigma \rightarrow bool\,)$$

The idea is that (I, N) represents a program that starts in a state satisfying I and repeatedly makes transitions allowed by N, under the assumption that if any action is enabled from some point on, it must be infinitely often executed. To formalize this idea, we need some definitions.

A *transition* is a triple $(s : \sigma,\ a : \alpha,\ s' : \sigma)$, representing a step of program execution. An *execution* is a pair $(xs : num \rightarrow \sigma,\ xa : num \rightarrow \alpha)$, representing an infinite history of program execution of the form:

$$xs(0) \xrightarrow{xa(0)} xs(1) \xrightarrow{xa(1)} xs(2) \xrightarrow{xa(2)} \cdots$$

Predicates on executions are called *temporal properties*. In the following, let $S, S' : \sigma \rightarrow bool$ be state predicates, $A : \alpha \rightarrow bool$ be an action predicate, $T : \sigma \times \alpha \times \sigma \rightarrow bool$ be a transition predicate, and P be a temporal property.

S (respectively, A or T) can be "coerced" into a temporal property $\mathsf{PS}(S)$ ($\mathsf{PA}(A)$ or $\mathsf{PT}(T)$) by evaluating it at the beginning of an execution:

$$\begin{aligned}
\mathsf{PS}(S)(xs, xa) &\triangleq S(xs(0))\\
\mathsf{PA}(A)(xs, xa) &\triangleq A(xa(0))\\
\mathsf{PT}(T)(xs, xa) &\triangleq T(xs(0), xa(0), xs(1))
\end{aligned}$$

The temporal modality $\Box$ (respectively, $\Diamond$) means "*always* (*sometime*) in the future":

$$\begin{aligned}
\Box(P)(x) &\triangleq \forall i : num\, .\, P(\mathsf{future}(i)(x))\\
\Diamond(P)(x) &\triangleq \exists i : num\, .\, P(\mathsf{future}(i)(x))
\end{aligned}$$

where x is an execution and "future" includes "now":

$$\mathsf{future}(i)(xs, xa) \triangleq (\, \lambda j\, .\, xs(i{+}j)\, ,\ \lambda j\, .\, xa(i{+}j)\,)$$

S *leads to* S' iff whenever S holds, S' holds then or later:

$$S \leadsto S' \triangleq \Box(\mathsf{PS}(S) \Rrightarrow \Diamond(\mathsf{PS}(S')))$$

T *enables* A at a state s iff T allows a transition labeled by some a in A from s:

$$\mathsf{Enable}(T)(A)(s) \triangleq \exists a :: A\, .\, \exists s'\, .\, T(s, a, s')$$

A is *fair* with respect to T iff T enabling A from some point on implies A being executed infinitely often:

$$\begin{aligned}
\mathsf{Fair}(T)(A) \triangleq\ & \Diamond(\Box(\mathsf{PS}(\mathsf{Enable}(T)(A)))) \Rrightarrow\\
& \Box(\Diamond(\mathsf{PA}(A)))
\end{aligned}$$

Now we are ready to define the *semantics* of a labeled transition system (I, N). An execution is *allowed* by (I, N) iff its initial state satisfies I, each of its transitions satisfies N, and each action $a : \alpha$ is fair with respect to N. The semantics of (I, N) is just the set of allowed executions:

$$\begin{aligned}
[\![(I, N)]\!] \triangleq\ & \mathsf{PS}(I) \mathbin{\wedge\!\!\!\wedge} \Box(\mathsf{PT}(N)) \mathbin{\wedge\!\!\!\wedge}\\
& \forall\!\!\!\forall a : \alpha\, .\, \mathsf{Fair}(N)(\{a\})
\end{aligned}$$

Note that we consider only *infinite* executions. This does not prevent us from modeling terminating programs, since a finite execution can be extended to an infinite one by repeating the terminal state. In fact, in Section 4, we will add to each of our programs a *stuttering* action—an action which is always enabled and whose execution leaves the state unchanged—to guarantee infinite executions. Note that, even with a stuttering action, the fairness assumption still ensures progress by ruling out those executions that settle into perpetual stuttering prematurely. The idea of using stuttering is borrowed from Lamport's TLA [22].

A labeled transition system Π *satisfies* a temporal property P iff every execution allowed by Π satisfies P:

$$\Pi \models P \triangleq \vdash\!\!(\, [\![\Pi]\!] \Rrightarrow P)$$

For proving statements of the form $\Pi \models P$, we have found some notions from Chandy and Misra's UNITY [5] to be useful and hence have formalized them in higher-order logic. Due to space limitations, we mention only two such notions here. We say S is an *invariant* of Π iff S is true at each initial state of Π and S is preserved by each transition of Π:

$$\begin{aligned}
&\mathsf{Invariant}(I, N)(S) \triangleq\\
&\quad (\forall s :: I\, .\, S(s)\,) \wedge\\
&\quad (\forall (s, a, t) :: N\, .\, S(s) \Rightarrow S(t)\,)
\end{aligned}$$

Then a simple inductive argument shows that:

$$\mathsf{Invariant}(\Pi)(S) \Rightarrow \Pi \models \Box(\mathsf{PS}(S)) \qquad (3)$$

We say S *ensures* S' in Π iff (a) if S holds but S' doesn't, then each transition of Π either keeps S true or makes S' true (termed "S *unless* S'" in [5]), and (b) there exists an action a of Π such that if S holds but S' doesn't, then a is enabled and executing a makes S' true:

$$\begin{aligned}
&\mathsf{Ensures}(I, N)(S)(S') \triangleq\\
&\quad (\forall (s, a, t) :: N\, .\, S(s) \wedge \neg S'(s) \Rightarrow S(t) \vee S'(t)\,) \wedge\\
&\quad (\exists a : \alpha\, .\, \forall s\, .\, S(s) \wedge \neg S'(s) \Rightarrow\\
&\qquad \mathsf{Enable}(N)(\{a\})(s) \wedge (\forall t\, .\, N(s, a, t) \Rightarrow S'(t))\,)
\end{aligned}$$

Ensures encapsulates our basic method for using the fairness assumption of a labeled transition system to prove its liveness properties:

$$\mathsf{Ensures}(\Pi)(S)(S') \Rightarrow \Pi \models S \leadsto S' \qquad (4)$$

The proof of (4) goes like this: Suppose $S \leadsto S'$ fails for some execution x of Π, i.e., S holds at some state s in x but S' never holds from s on in x. Then an inductive argument using the first clause of Ensures shows that S always holds from s on in x. Then the second clause of Ensures implies that there is an action a that is always enabled but never executed from s on in x, which violates the fairness assumption of Π.

3.3 Simulation of Programs

Consider the following two labeled transition systems:

$$\begin{array}{rcl} \Pi^\flat & = & (I^\flat : \sigma^\flat \rightarrow \mathit{bool},\ N^\flat : \sigma^\flat \times \alpha \times \sigma^\flat \rightarrow \mathit{bool}) \\ \Pi^\sharp & = & (I^\sharp : \sigma^\sharp \rightarrow \mathit{bool},\ N^\sharp : \sigma^\sharp \times \alpha \times \sigma^\sharp \rightarrow \mathit{bool}) \end{array}$$

which share the same type α of actions. $\Pi^\flat$ represents a "concrete" program with a "concrete" state space $\sigma^\flat$ and $\Pi^\sharp$ represents an "abstract" program with an "abstract" state space $\sigma^\sharp$. We say that $\Pi^\flat$ can be *simulated* by $\Pi^\sharp$ via a *joint invariant* $J : \sigma^\flat \rightarrow \sigma^\sharp \rightarrow \mathit{bool}$, denoted $\mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp)$, iff all of the following are true:

S1 $\forall s^\flat : \sigma^\flat .\ I^\flat(s^\flat) \Rightarrow$
$\quad \exists s^\sharp : \sigma^\sharp .\ I^\sharp(s^\sharp) \wedge J(s^\flat)(s^\sharp)$

S2 $\forall s^\flat : \sigma^\flat .\ \forall s^\sharp : \sigma^\sharp .\ J(s^\flat)(s^\sharp) \Rightarrow$
$\quad \forall a : \alpha .\ \forall t^\flat : \sigma^\flat .\ N^\flat(s^\flat, a, t^\flat) \Rightarrow$
$\quad\quad \exists t^\sharp : \sigma^\sharp .\ N^\sharp(s^\sharp, a, t^\sharp) \wedge J(t^\flat)(t^\sharp)$

S3 $\forall s^\flat : \sigma^\flat .\ \forall s^\sharp : \sigma^\sharp .\ J(s^\flat)(s^\sharp) \Rightarrow$
$\quad \forall a : \alpha .\ \mathsf{Enable}(N^\sharp)(\{a\})(s^\sharp) \Rightarrow$
$\quad\quad \mathsf{Enable}(N^\flat)(\{a\})(s^\flat)$

If $\mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp)$, then for any execution $x^\flat$ allowed by $\Pi^\flat$, there exists an execution $x^\sharp$ allowed by $\Pi^\sharp$ such that each pair of corresponding states in $x^\flat$ and $x^\sharp$ satisfies the joint invariant J:

$$\begin{array}{rl} \mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp) \ \Rightarrow & \forall x^\flat :: [\![\Pi^\flat]\!] . \\ & \exists x^\sharp :: [\![\Pi^\sharp]\!] .\ \Box J\,(x^\flat)(x^\sharp) \end{array} \tag{5}$$

where we overload the symbol $\Box$ and define:

$$\Box J\,(xs^\flat, xa^\flat)(xs^\sharp, xa^\sharp) \ \triangleq\ \forall n .\ J(xs^\flat(n))(xs^\sharp(n))$$

Theorem (5) is proved in two steps. First, given any execution $x^\flat = (xs^\flat, xa^\flat)$ allowed by $\Pi^\flat$, an execution $x^\sharp = (xs^\sharp, xa^\sharp)$ is constructed inductively using **S1** and **S2** such that:

$$\begin{array}{l} (\,\mathsf{PS}(I^\sharp) \mathbin{\wedge\!\!\!\wedge} \Box(\mathsf{PT}(N^\sharp))\,)(x^\sharp) \wedge \\ \Box J\,(x^\flat)(x^\sharp) \wedge (\forall n .\ xa^\flat(n) = xa^\sharp(n)\,) \end{array} \tag{6}$$

Second, **S3** is used to show that for any two executions $x^\flat$ and $x^\sharp$ related by the second line of (6), if $x^\flat$ satisfies the fairness assumption of $\Pi^\flat$, then $x^\sharp$ satisfies the fairness assumption of $\Pi^\sharp$.

Our notion of simulation generalizes Milner simulation [29], which essentially consists of **S1** and **S2**; we added **S3** to deal with fairness assumptions. Note that, in (6), each pair of corresponding actions in $x^\flat$ and $x^\sharp$, $xa^\flat(n)$ and $xa^\sharp(n)$, must be identical. This requirement can be relaxed by allowing $\Pi^\flat$ and $\Pi^\sharp$ to have different types of actions $\alpha^\flat$ and $\alpha^\sharp$ and using an "action invariant" $K : \alpha^\flat \rightarrow \alpha^\sharp \rightarrow \mathit{bool}$ to relate $xa^\flat(n)$ and $xa^\sharp(n)$. Though not needed in this paper, such a generalization is useful in reasoning about more complex distributed algorithms.

3.4 Translation of Properties

Let ξ and ζ be two arbitrary types and $R : \xi \rightarrow \zeta \rightarrow \mathit{bool}$ be a relation. We say that a predicate $X : \xi \rightarrow \mathit{bool}$ can be *translated* into a predicate $Y : \zeta \rightarrow \mathit{bool}$ via the relation R iff for any $x : \xi$ and $y : \zeta$ related by R, if x satisfies X, then y satisfies Y:

$$\begin{array}{rl} X \{\!\!\{R\}\!\!\} Y \ \triangleq & \forall x : \xi .\ \forall y : \zeta . \\ & R(x)(y) \Rightarrow (\,X(x) \Rightarrow Y(y)\,) \end{array} \tag{7}$$

The $X \{\!\!\{R\}\!\!\} Y$ notation is inspired by the (original) notation for Hoare triples [19]; indeed, if R is the relational semantics of a command, then $X \{\!\!\{R\}\!\!\} Y$ is a Hoare triple. For later use, we define the *inverse* $R^{-1} : \zeta \rightarrow \xi \rightarrow \mathit{bool}$ by $R^{-1}(y)(x) \triangleq R(x)(y)$.

For any labeled transition systems $\Pi^\flat$ and $\Pi^\sharp$ with a common type of actions and any temporal properties $P^\flat$ and $P^\sharp$ of the appropriate types, it easily follows from (5) and (7) that:

$$\begin{array}{l} \mathsf{Sim}(J)(\Pi^\flat)(\Pi^\sharp) \wedge P^\sharp \{\!\!\{\Box(J^{-1})\}\!\!\} P^\flat \\ \quad \Rightarrow\ (\,\Pi^\sharp \models P^\sharp \Rightarrow \Pi^\flat \models P^\flat\,) \end{array} \tag{8}$$

which is the precise statement of the Fundamental Theorem depicted in (1).

Theorem (8) would be virtually useless if translation relations of the form $P^\sharp \{\!\!\{\Box(J^{-1})\}\!\!\} P^\flat$ could only be proved by appealing to Definition (7) directly, since $P^\flat$ and $P^\sharp$ can be complex temporal properties. Fortunately, there are *Reduction Lemmas* (listed below) that allow one to reduce the proof of a translation relation between two complex properties *of the same form* to the proofs of translation relations between their corresponding constituents. In the following, S and T are state predicates, P and Q are temporal properties, and X, Y, X_i's, and Y_i's are arbitrary predicates, where i ranges over an arbitrary type.

$$Y \{\!\!\{R^{-1}\}\!\!\} X \ \Rightarrow\ (\neg X) \{\!\!\{R\}\!\!\} (\neg Y) \tag{9}$$

$$\begin{array}{l} X_1 \{\!\!\{R\}\!\!\} Y_1 \wedge X_2 \{\!\!\{R\}\!\!\} Y_2 \\ \quad \Rightarrow\ (X_1 \mathbin{\wedge\!\!\!\wedge} X_2) \{\!\!\{R\}\!\!\} (Y_1 \mathbin{\wedge\!\!\!\wedge} Y_2) \end{array} \tag{10}$$

$$\begin{array}{l} X_1 \{\!\!\{R\}\!\!\} Y_1 \wedge X_2 \{\!\!\{R\}\!\!\} Y_2 \\ \quad \Rightarrow\ (X_1 \mathbin{\vee\!\!\!\vee} X_2) \{\!\!\{R\}\!\!\} (Y_1 \mathbin{\vee\!\!\!\vee} Y_2) \end{array} \tag{11}$$

$$\begin{array}{l} Y_1 \{\!\!\{R^{-1}\}\!\!\} X_1 \wedge X_2 \{\!\!\{R\}\!\!\} Y_2 \\ \quad \Rightarrow\ (X_1 \Rrightarrow X_2) \{\!\!\{R\}\!\!\} (Y_1 \Rrightarrow Y_2) \end{array} \tag{12}$$

$$(\forall i.\, X_i \{\!\!\{R\}\!\!\} Y_i) \;\Rightarrow\; (\mathbb{V} i.\, X_i) \{\!\!\{R\}\!\!\} (\mathbb{V} i.\, Y_i) \quad (13)$$

$$(\forall i.\, X_i \{\!\!\{R\}\!\!\} Y_i) \;\Rightarrow\; (\exists i.\, X_i) \{\!\!\{R\}\!\!\} (\exists i.\, Y_i) \quad (14)$$

$$S \{\!\!\{J\}\!\!\} T \;\Rightarrow\; (\mathsf{PS}(S)) \{\!\!\{\Box J\}\!\!\} (\mathsf{PS}(T)) \quad (15)$$

$$P \{\!\!\{\Box J\}\!\!\} Q \;\Rightarrow\; (\Box P) \{\!\!\{\Box J\}\!\!\} (\Box Q) \quad (16)$$

$$P \{\!\!\{\Box J\}\!\!\} Q \;\Rightarrow\; (\Diamond P) \{\!\!\{\Box J\}\!\!\} (\Diamond Q) \quad (17)$$

None of (9)–(17) is hard to prove.

As an example of the application of the Reduction Lemmas, we can use the definition:

$$S \leadsto S' \;\triangleq\; \Box(\mathsf{PS}(S) \Rightarrow \Diamond(\mathsf{PS}(S')))$$

and Lemmas (16), (12), (17), and (15) (in that order) to prove:

$$\begin{aligned} & T_1 \{\!\!\{J^{-1}\}\!\!\} S_1 \wedge S_2 \{\!\!\{J\}\!\!\} T_2 \\ & \quad \Rightarrow\; (S_1 \leadsto S_2) \{\!\!\{\Box J\}\!\!\} (T_1 \leadsto T_2) \end{aligned} \quad (18)$$

So the task of proving the second line of (18) can be reduced to that of proving the first line of (18), which involves no temporal reasoning. In general, if P and Q are two temporal properties *of the same form* whose primitive constituents are state predicates, then the task of proving $P \{\!\!\{\Box J\}\!\!\} Q$ can be reduced to proving translation relations between state predicates and hence will not involve any temporal reasoning.

3.5 Graph Theory

In this paper, every graph is *undirected* and *finite* and may have at most one edge between two nodes. Hence an edge connecting two nodes p and q can be identified with the set $\{p, q\}$. Let G be a graph. We say that two nodes p and q are *adjacent* in G, denoted $\mathsf{Adjacent}(G)(p)(q)$, iff $\{p, q\}$ is an edge in G. Note that by the identification of sets with characteristic predicates (Section 3.1.1), the set of adjacent nodes of p in G is simply $\mathsf{Adjacent}(G)(p)$. A *link* of G is a pair (p, q) such that $\mathsf{Adjacent}(G)(p)(q)$. The sets of nodes, edges, and links of G are denoted by $\mathsf{Node}(G)$, $\mathsf{Edge}(G)$, and $\mathsf{Link}(G)$ respectively.

A *tree* is a connected and acyclic graph. A crucial property of trees is that there is exactly one path between any two nodes in a tree, where a path can pass through an edge at most once. Let T be a tree and r (for "root"), n, and p be nodes in T. We say p is a *parent* of n with respect to T and r, denoted $\mathsf{Parent}(T)(r)(n)(p)$, iff $\mathsf{Adjacent}(T)(n)(p)$, $n \neq r$, and the path from n to r passes through p. Note that each $n \neq r$ in T has a unique parent, which is denoted by $\mathsf{TheParent}(T)(r)(n)$. In the sequel we will drop the argument r since it will be clear from context. A graph T is a *subtree* of another graph G iff T is both a subgraph of G and a tree; the set of subtrees of G is denoted by $\mathsf{Subtree}(G)$.

4 THE EXAMPLE

4.1 The Distributed Algorithm PIF

Let G be a *connected* graph whose nodes represent autonomous processors and whose links represent communication channels via which the processors can send messages to each other. Let r be a distinguished node in G called the root and i be a piece of information initially residing at r. The purpose of PIF is to propagate i to all other nodes in G and to notify r after i has indeed reached every node in G. PIF operates as follows. At the beginning, the root spontaneously wakes up and sends i to each of its neighbors. When a non-root node receives i for the first time, it marks as its parent the node from which it receives the first i and sends i to each of its neighbors except its parent. When a node has received i from each of its neighbors, it stops and, if it is not the root, sends i to its parent before stopping. PIF terminates when the root stops.

Since we are interested in proving the correctness of PIF for *any* connected network G, node r in G, and information i, the tuple (G, r, i) is an argument of every constant defined in this section. However, to simplify notation, it will not be explicitly mentioned.

We formalize PIF as a labeled transition system:

$$\begin{aligned} \mathsf{Prog}^D \;\triangleq\; (\,&\mathsf{Init}^D \;: sta^D \to bool, \\ &\mathsf{Next}^D : sta^D \times act \times sta^D \to bool\,) \end{aligned}$$

where the superscript ^{D} indicates that this is the *distributed view* of PIF; in the next subsection, the *event view* of PIF will be given the superscript ^{E}. We now describe what sta^D, $Init^D$, and Next^D, and act are.

PIF has four program variables at each node n in G and a message queue over each link l in G:

$pc(n)$	:	pc	(* program counter *)
$inf(n)$	:	ι	(* information *)
$par(n)$	:	ν	(* parent *)
$cnt(n)$	:	num	(* counter *)
$mq(l)$	:	$(\iota)list$	(* message queue *)

where ι is the type of information, ν is the type of nodes, $(\iota)list$ is the type of lists with elements from ι, and pc is the type of program counters defined by:

$$pc \;::=\; \texttt{Idle} \mid \texttt{Busy} \mid \texttt{Done}$$

Therefore, the type sta^D of states of Prog^D is:

$$(\nu \to pc) \times (\nu \to \iota) \times (\nu \to \nu) \times (\nu \to num) \times (\nu \times \nu \to (\iota)list)$$

Initially, all we know is that every program counter is `Idle` and every message queue is empty:

$$\begin{aligned} \mathsf{Init}^D(ds) \;\triangleq\; &(\forall n :: \mathsf{Node}(G)\,.\, pc(n) = \texttt{Idle}\,) \wedge \\ &(\forall l :: \mathsf{Link}(G)\,.\, mq(l) = [\;]\,) \end{aligned}$$

where ds abbreviates $(pc, inf, par, cnt, mq) : sta^D$.

The atomic actions of PIF are listed below, where the ADA-like pseudo-code has the obvious meaning. For instance, an action of the form:

n receive m from p when b begin c end

is enabled when n and p are adjacent nodes, m is the message at the head of the message queue from p to n, and the condition b is true. The action is executed by removing m from the message queue and then sequentially executing the command c, all in one *atomic* step.

```
action rootBegin($n$)
when $(n = r) \wedge (pc(n) = \mathtt{Idle})$
begin
  $pc(n) := \mathtt{Busy}$ ;   $inf(n) := i$ ;
  let $Q = \mathsf{Adjacent}(G)(n)$ in
    $cnt(n) := |Q|$ ;   for $q$ in $Q$ do $n$ send $i$ to $q$ ;
end

action nodeBegin($n$)($p$)
$n$ receive $j$ from $p$ when $(pc(n) = \mathtt{Idle})$
begin
  $pc(n) := \mathtt{Busy}$ ;   $inf(n) := j$ ;   $par(n) := p$ ;
  let $Q = \mathsf{Adjacent}(G)(n) \setminus \{p\}$ in
    $cnt(n) := |Q|$ ;   for $q$ in $Q$ do $n$ send $j$ to $q$ ;
end

action adjEnd($n$)($p$)
$n$ receive $j$ from $p$ when $(pc(n) \neq \mathtt{Idle})$
begin
  $cnt(n) := cnt(n) - 1$ ;
end

action nodeEnd($n$)
when $(pc(n) = \mathtt{Busy}) \wedge (cnt(n) = 0)$
begin
  $pc(n) := \mathtt{Done}$ ;
  if $(n \neq r)$ then $n$ send $inf(n)$ to $par(n)$ ;
end

action Stutter
when T begin end
```

It is straightforward to translate the above pseudo-code into the higher-order logic formula Next^D, but we will not do so here due to space limitations. The type *act* of action names of Prog^D is defined by:

$$act \ ::= \ \mathtt{rootBegin}(\nu) \mid \mathtt{nodeBegin}(\nu)(\nu) \mid \mathtt{adjEnd}(\nu)(\nu) \mid \mathtt{nodeEnd}(\nu) \mid \mathtt{Stutter}$$

Our goal is to prove that Prog^D satisfies the following three temporal properties:

$$\mathsf{Prog}^D \models \Box(\mathsf{PS}(\mathsf{Done}^D \Rightarrow \mathsf{Final}^D)) \qquad (19)$$
$$\mathsf{Prog}^D \models \Box(\mathsf{PS}(\mathsf{Done}^D) \Rightarrow \Box(\mathsf{PS}(\mathsf{Done}^D))) \qquad (20)$$
$$\mathsf{Prog}^D \models \Diamond(\mathsf{PS}(\mathsf{Done}^D)) \qquad (21)$$

Statement (19) says that whenever Prog^D terminates, its state is in a desired condition (viz., the partial correctness of Prog^D), where Done^D signifies termination:

$$\mathsf{Done}^D(ds) \ \triangleq \ (pc(r) = \mathtt{Done})$$

and Final^D describes the desired terminal states:

$$\mathsf{Final}^D(ds) \ \triangleq \ (\forall n :: \mathsf{Node}(G) .\ inf(n) = i)$$

Statement (20) says that once Prog^D terminates, it stays that way forever. Statement (21) says that Prog^D must eventually terminate.

4.2 Event-and-Causality View of PIF

The *event view* of PIF is an abstract program:

$$\mathsf{Prog}^E \ \triangleq \ (\mathsf{Init}^E \ : sta^E \to bool,\ \mathsf{Next}^E : sta^E \times act \times sta^E \to bool)$$

which is an operational representation of the events and causality relations that PIF can generate. Note that Prog^E has the same type *act* of action names as Prog^D.

Prog^E has only two program variables: T, which records the tree induced by Busy nodes and their parent pointers, and occ, which records the set of events that have occurred so far. So the type sta^E of states of Prog^E is $graph \times (event \to bool)$, where *graph* is the type of graphs and *event* is the type of events of Prog^E defined by:[2]

$$event \ ::= \ \mathtt{adjEnd}(\nu)(\nu) \mid \mathtt{nodeEnd}(\nu)$$

Note that there is no need to record the occurrences of $\mathtt{rootBegin}(n)$ and $\mathtt{nodeBegin}(n)(p)$, since they can be deduced from T, or $\mathtt{Stutter}$, since they have no effect anyway. Initially, T is empty and no event has occurred:

$$\mathsf{Init}^E(es) \ \triangleq \ (T = \emptyset) \wedge (occ = \emptyset)$$

where es abbreviates $(T, occ) : sta^E$.

The causality relations can be easily expressed in terms of T and occ by specifying the (immediate) cause $\mathsf{Cause}^E(ev)$ for each $ev : event$, as follows:

$$\begin{aligned}&\mathsf{Cause}^E(\mathtt{adjEnd}(n)(p))(es) \triangleq \\ &\quad \mathsf{Node}(T)(n) \wedge \mathsf{Node}(T)(p) \wedge \\ &\quad ((\mathsf{Parent}(T)(p)(n) \wedge occ(\mathtt{nodeEnd}(p))) \vee \\ &\quad\ (\mathsf{Adjacent}(G)(n)(p) \wedge \neg\mathsf{Adjacent}(T)(n)(p)))\end{aligned}$$

$$\begin{aligned}&\mathsf{Cause}^E(\mathtt{nodeEnd}(n))(es) \triangleq \\ &\quad \mathsf{Node}(T)(n) \wedge \\ &\quad \forall p :: \mathsf{Adjacent}(G)(n) . \\ &\quad\ \ \mathsf{Parent}(T)(n)(p) \vee occ(\mathtt{adjEnd}(n)(p))\end{aligned}$$

[2]Here we have defined *event* to be a subtype of *act*, which is actually not allowed in HOL. But, to simplify notation, we will pretend we could do so.

A `rootBegin`(n) action is enabled when n is the root but is not in T, and its effect is to add n to T but keep occ unchanged (where es' abbreviates $(T', occ') : sta^E$):

$$\begin{aligned}&\mathsf{Next}^E(es, \texttt{rootBegin}(n), es') \triangleq\\&\quad (n = r) \wedge \neg\mathsf{Node}(T)(n) \wedge\\&\quad (T' = T \cup \{n\}) \wedge (occ' = occ)\end{aligned}$$

A `nodeBegin`$(n)(p)$ action is enabled when p is in T but n is not in T, and its effect is to add the edge $\{n,p\}$ to T but keep occ unchanged:

$$\begin{aligned}&\mathsf{Next}^E(es, \texttt{nodeBegin}(n)(p), es') \triangleq\\&\quad \mathsf{Node}(T)(p) \wedge \neg\mathsf{Node}(T)(n) \wedge\\&\quad (T' = T \cup \{\{n,p\}\}) \wedge (occ' = occ)\end{aligned}$$

An event $ev : event$ is enabled when the cause of ev is true but ev has not yet occurred, and its effect is to add ev to occ but keep T unchanged:

$$\begin{aligned}&\mathsf{Next}^E(es, ev, es') \triangleq\\&\quad \mathsf{Cause}^E(ev)(es) \wedge \neg occ(ev) \wedge\\&\quad (T' = T) \wedge (occ' = occ \cup \{ev\})\end{aligned}$$

The `Stutter` action has the obvious meaning:

$$\begin{aligned}&\mathsf{Next}^E(es, \texttt{Stutter}, es') \triangleq\\&\quad (T' = T) \wedge (occ' = occ)\end{aligned}$$

The event view Prog^E has a very simple invariant which says that (a) T is a subtree of G, (b) T either is empty or contains the root, and (c) if an event has occurred, then its cause must be true:

$$\begin{aligned}&\mathsf{Inv}^E(es) \triangleq\\&\quad \mathsf{Subtree}(G)(T) \wedge (\mathsf{Node}(T) = \emptyset \vee \mathsf{Node}(T)(r)) \wedge\\&\quad (\forall\, ev : event .\ occ(ev) \Rightarrow \mathsf{Cause}^E(ev)(es))\end{aligned}$$

The invariance of Inv^E can be easily proved using (3):

$$\mathsf{Prog}^E \ \models\ \Box(\mathsf{PS}(\mathsf{Inv}^E)) \qquad (22)$$

The termination of Prog^E is signified by the occurrence of `nodeEnd`(r):

$$\mathsf{Done}^E(es) \ \triangleq\ occ(\texttt{nodeEnd}(r))$$

Clearly, once an event is added to occ, it stays in occ forever. In particular, once Prog^E terminates, it stays that way forever:

$$\mathsf{Prog}^E \ \models\ \Box(\mathsf{PS}(\mathsf{Done}^E) \Rightarrow \Box(\mathsf{PS}(\mathsf{Done}^E))) \qquad (23)$$

The basic liveness properties of Prog^E are:[3]

$$\mathsf{Prog}^E \ \models\ (|\mathsf{Node}(T)| = n) \leadsto (|\mathsf{Node}(T)| > n) \qquad (24)$$

for each $n : num$, which says that T must get bigger and bigger, and:

$$\mathsf{Prog}^E \ \models\ \mathsf{Cause}^E(ev)(occ) \leadsto occ(ev) \qquad (25)$$

for each $ev : event$, which says that once the cause of an event becomes true, the event must eventually occur. Both (24) and (25) can be proved using (4). Since T never shrinks, (24) guarantees that T must become a spanning tree of G from some point on:

$$\mathsf{Prog}^E \ \models\ \Diamond\Box(\mathsf{Node}(T) = \mathsf{Node}(G)) \qquad (26)$$

Once T has become a spanning tree of G, repeated application of (25) from the leaves to the root of T shows that `nodeEnd`(r) must eventually occur, i.e., Prog^E must eventually terminate:

$$\mathsf{Prog}^E \ \models\ \Diamond(\mathsf{PS}(\mathsf{Done}^E)) \qquad (27)$$

4.3 Relating the Two Views of PIF

The following is a joint invariant $\mathsf{Inv}^{D,E} : sta^D \to sta^E \to bool$ such that $\mathsf{Sim}(\mathsf{Inv}^{D,E})(\mathsf{Prog}^D)(\mathsf{Prog}^E)$:

$$\begin{aligned}&\mathsf{Inv}^{D,E}(ds)(es) \triangleq\\&\quad \mathsf{Inv}^E(es) \wedge (\forall n :: \mathsf{Node}(G) .\ \mathsf{Inv}^{D,E}_N(n)(ds)(es))\\&\quad\quad \wedge (\forall l :: \mathsf{Link}(G) .\ \mathsf{Inv}^{D,E}_L(l)(ds)(es))\end{aligned}$$

where Inv^E is the invariant of Prog^E and $\mathsf{Inv}^{D,E}_N(n)$ (respectively, $\mathsf{Inv}^{D,E}_L(l)$) essentially[4] expresses the local state of node n (link l) in Prog^D in terms of the state of Prog^E:

$$\begin{aligned}&\mathsf{Inv}^{D,E}_N(n)(ds)(es) \triangleq\\&\quad (pc(n) =\\&\quad\ \ \textbf{if } occ(\texttt{nodeEnd}(n)) \textbf{ then } \texttt{Done} \textbf{ else}\\&\quad\ \ \textbf{if } \mathsf{Node}(T)(n) \textbf{ then } \texttt{Busy} \textbf{ else } \texttt{Idle}) \wedge\\&\quad (\mathsf{Node}(T)(n) \Rightarrow\\&\quad\ \ (inf(n) = i) \wedge\\&\quad\ \ ((n \neq r) \Rightarrow (par(n) = \mathsf{TheParent}(T)(n))) \wedge\\&\quad\ \ (cnt(n) = |\{ q \mid \mathsf{Adjacent}(G)(n)(q) \wedge\\&\quad\quad\quad\quad \neg\mathsf{Parent}(T)(n)(q) \wedge\\&\quad\quad\quad\quad \neg occ(\texttt{adjEnd}(n)(q)) \}|))\end{aligned}$$

$$\begin{aligned}&\mathsf{Inv}^{D,E}_L(p,q)(ds)(es) \triangleq\\&\quad (mq(p,q) =\\&\quad\ \ \textbf{if } \mathsf{Node}(T)(p) \wedge \neg occ(\texttt{adjEnd}(q)(p)) \wedge\\&\quad\quad ((\mathsf{Parent}(T)(p)(q) \wedge occ(\texttt{nodeEnd}(p))) \vee\\&\quad\quad\ (\mathsf{Adjacent}(G)(p)(q) \wedge \neg\mathsf{Adjacent}(T)(p)(q)))\\&\quad\ \ \textbf{then } [i] \textbf{ else } [\,])\end{aligned}$$

The proof of $\mathsf{Sim}(\mathsf{Inv}^{D,E})(\mathsf{Prog}^D)(\mathsf{Prog}^E)$ (i.e., **S1–S3**) is omitted due to space limitations.

To translate the properties (22), (23), and (27) of Prog^E into the desired properties (19), (20), and (21)

[3] We are abusing notation in (24): by $(|\mathsf{Node}(T)| = n)$ we really mean a predicate C such that $C(es) = (|\mathsf{Node}(T)| = n)$; similar remarks apply to (25) and (26) as well.

[4] A truly functional relation from es to ds cannot be used because most program variables of Prog^D have unspecified initial values.

of Prog^D, we need to prove the following translation relations:

$$(\Box(\mathsf{PS}(\mathsf{Inv}^E)))\ \{\!\!\{\Box(\mathsf{Inv}^{D,E})^{-1}\}\!\!\}\ (\Box(\mathsf{PS}(\mathsf{Done}^D \Rightarrow \mathsf{Final}^D)))$$

$$(\Box(\mathsf{PS}(\mathsf{Done}^E) \Rightarrow \Box(\mathsf{PS}(\mathsf{Done}^E))))\ \{\!\!\{\Box(\mathsf{Inv}^{D,E})^{-1}\}\!\!\}\ (\Box(\mathsf{PS}(\mathsf{Done}^D) \Rightarrow \Box(\mathsf{PS}(\mathsf{Done}^D))))$$

$$(\Diamond(\mathsf{PS}(\mathsf{Done}^E)))\ \{\!\!\{\Box(\mathsf{Inv}^{D,E})^{-1}\}\!\!\}\ (\Diamond(\mathsf{PS}(\mathsf{Done}^D)))$$

which can be reduced, using the Reduction Lemmas in the same manner as illustrated in the proof of (18), to the following three proof obligations:

$$(\mathsf{Inv}^E)\ \{\!\!\{(\mathsf{Inv}^{D,E})^{-1}\}\!\!\}\ (\mathsf{Done}^D \Rightarrow \mathsf{Final}^D)$$
$$(\mathsf{Done}^E)\ \{\!\!\{(\mathsf{Inv}^{D,E})^{-1}\}\!\!\}\ (\mathsf{Done}^D)$$
$$(\mathsf{Done}^D)\ \{\!\!\{\mathsf{Inv}^{D,E}\}\!\!\}\ (\mathsf{Done}^E)$$

all of which can be proved using the definition of translation (7) directly. Again, the proof is omitted due to space limitations.

Finally, the Fundamental Theorem (8) can be used to conclude properties (19), (20), and (21) of Prog^D from properties (22), (23), and (27) of Prog^E. The correctness proof of PIF is now complete. Note that all temporal reasoning involved is carried out in the last subsection in terms of the event view Prog^E, which is more abstract and easier to reason about than Prog^D. Note also that once the notions of events and causality are understood, it is quite easy to write down the joint invariant $\mathsf{Inv}^{D,E}$.

5 RELATED WORK

The idea of using simulation to relate a program and its abstraction goes back at least as far as Milner [29]. For a comprehensive survey of various simulation techniques, see [25]. Note, however, that the simulation techniques surveyed in [25] are based on models in which there is a distinction between "external" and "internal" components of states or actions, and that they are designed to show the *containment* of the set of external behaviors of a concrete program in that of an abstract program. Our approach does not distinguish between external and internal components of states or actions; instead, we use simulation to establish a relation between concrete and abstract executions and then translate properties of the latter into those of the former via that relation. In addition to this paper, the mechanization of simulation techniques is also treated in [7, 24, 31, 35].

Hoare triples were introduced in [19]. But, as far as we know, using them as a relational formulation of translation of properties is new. Also new is the formulation and use of the Reduction Lemmas.

The notions of events and causality are not new [21, 30]. Indeed, a variety of *non*-interleaving models of concurrency have been proposed in the literature, such as partial orders of events [21, 34], event structures [30], Mazurkiewicz traces [26], and asynchronous transition systems [34]; see [34] for a lucid exposition of the relationships between these models. While a great deal of theoretical investigation has been conducted, none of these works addresses the practical problem of how to verify realistic distributed algorithms using these models. We hope that this paper is a small step in that direction. Also, it should be noted that although we use the intuition about events and causality as an *informal* guide for structuring proofs, the *formal* foundation of our methodology is still the interleaving view of concurrency.

Section 3.2 is influenced by Chandy and Misra's UNITY [5] and Lamport's TLA [22]. But we borrow only those notions that are useful to us, and we freely adapt them to suit our own purpose. For more faithful embeddings of UNITY and TLA in HOL, see [1, 23].

Goldschlag [17] embeds UNITY in the Boyer-Moore prover [4] and uses the combination to verify a distributed minimum-finding algorithm for trees, based on a detailed hand proof by Lamport. Since the quantifier-free first-order logic used in the Boyer-Moore prover is less expressive than higher-order logic, his formalization is more subtle and indirect than what is possible in higher-order logic. Also, he does not develop a theory of general graphs, but uses nested lists to represent trees.

Engberg, Grønning, and Lamport [14] implement a translator of TLA into LP (the Larch Prover) [16] and use the combination to verify a distributed algorithm for computing the distances of all nodes to a distinguished node in a network. Consequently they must face the difficult problem of how to interface two different logics in a meaningful and consistent way, which we avoid by working in a single logic. Also, they do not prove the axioms and inference rules of TLA and the properties of data structures used in their proof, which they either keep as assumptions or assert as axioms. Indeed, there does not seem to be a clear distinction in LP between definitions and arbitrary axioms.

6 POSSIBLE EXTENSIONS

This work can be extended in at least two directions. First, one can investigate the applicability of the reasoning infrastructure (Section 3) and the event-and-causality-based approach (Section 4) to distributed algorithms more complex than PIF. In fact, we believe [10] that our methodology can be scaled up to verify algorithms as complex as the distributed minimum spanning tree algorithm of Gallager, Humblet, and Spira [15]. But, of course, only actual experience will be able to tell whether or not this belief is correct.

Second, though it is unlikely that the verification of distributed algorithms can ever be completely automated, the programmability of HOL allows one to write customized proof procedures to automate many tasks in proofs. At present we already have procedures for the uniform lifting of HOL proof procedures to handle lifted validity [6], for the automatic derivation of the enabling conditions of actions, and for the chasing of causality chains. We expect more automation be achieved as more experience is gained and more common patterns in reasoning are observed.

Acknowledgements

The author is grateful to Professors Eli Gafni and David Martin for their guidance, to Michael Gordon for his encouragement, to Flemming Andersen, Peter Homeier, Sara Kalvala, Stott Parker, Konrad Slind, Joakim von Wright, and the anonymous referees for their helpful comments, and to HOL users around the world for sustaining a lively and friendly user community.

REFERENCES

[1] F. Andersen, K.D. Petersen, and J.S. Pettersson, "Program Verification Using HOL-UNITY", in [20], pp. 1–15.

[2] J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (Ed.), *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1988.

[3] G.v. Bochmann and D.K. Probst (Ed.), *Computer-Aided Verification, 4th Int. Workshop*, LNCS 663, Springer-Verlag, 1992.

[4] Robert S. Boyer and J Strother Moore, *A Computational Logic*, Academic Press, 1979.

[5] K. Mani Chandy and Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[6] Ching-Tsun Chou, "A Sequent Formulation of a Logic of Predicates in HOL", in [12], pp. 71–80.

[7] Ching-Tsun Chou, "Predicates, Temporal Logic, and Simulations", in [20], pp. 310–323.

[8] Ching-Tsun Chou, "A Formal Theory of Undirected Graphs in Higher-Order Logic", in [28], pp. 144–157.

[9] Ching-Tsun Chou, "Mechanical Verification of Distributed Algorithms in Higher-Order Logic", in [28], pp. 158–176.

[10] Ching-Tsun Chou and Eli Gafni, "Understanding and Verifying Distributed Algorithms Using Stratified Decomposition", *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pp. 44–65, Aug. 1988.

[11] Alonzo Church, "A Formulation of the Simple Theory of Types", in *Journal of Symbolic Logic*, Vol. 5, pp. 56–68, 1940.

[12] L.J.M. Claesen and M.J.C. Gordon (Ed.), *Higher Order Logic Theorem Proving and Its Applications, 5th International Workshop*, IFIP Transactions A-20, North-Holland, 1992.

[13] C. Courcoubetis (Ed.), *Computer-Aided Verification, 5th Int. Workshop*, LNCS 697, Springer-Verlag, 1993.

[14] Urban Engberg, Peter Grønning, and Leslie Lamport, "Mechanical Verification of Concurrent Systems with TLA", in [3], pp. 44–55.

[15] R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 1, pp. 66–77, Jan. 1983.

[16] S.J. Garland and J.V. Guttag, "A Guide to LP, the Larch Prover", Research Report 82, DEC Systems Research Center, Dec. 1991.

[17] D.M. Goldschlag, "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover", *IEEE Trans. on Software Engineering*, Vol. 16, No. 9, pp. 1005–1023, Sep. 1990.

[18] M.J.C. Gordon and T.F. Melham (Ed.), *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.

[19] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, Vol. 12, No. 10, pp. 576–583, Oct. 1969.

[20] J.J. Joyce and C.-J.H. Seger (Ed.), *Higher Order Logic Theorem Proving and Its Applications, 6th International Workshop*, LNCS 780, Springer-Verlag, 1993.

[21] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, Jul. 1978.

[22] Leslie Lamport, "The Temporal Logic of Actions", *ACM Trans. on Programming Languages and Systems*, Vol. 16, No. 3, pp. 872–923, May 1994.

[23] Thomas Långbacka, "A HOL Formalisation of the Temporal Logic of Actions", in [28], pp. 332–345.

[24] Paul Loewenstein, "A Formal Theory of Simulations between Infinite Automata", in [12], pp. 227–246.

[25] Nancy A. Lynch and Frits W. Vaandrager, "Forward and Backward Simulations, Part I: Untimed Systems", CWI Report CS-R9313, March 1993. (To appear in *Information and Computation*.)

[26] Antoni Mazurkiewicz, "Basic Notions of Trace Theory", in [2], pp. 285–363.

[27] Thomas F. Melham, "Automating Recursive Type Definitions in Higher-Order Logic", pp. 341–386 of G. Birtwistle and P.A. Subrahmanyam (Ed.), *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989.

[28] T.F. Melham and J. Camilleri (Ed.), *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, LNCS 859, Springer-Verlag, 1994.

[29] Robin Milner, "An Algebraic Definition of Simulation between Programs", *Proc. of the 2nd Int. Joint Conf. on Artificial Intelligence*, pp. 481–489, 1971.

[30] M. Nielsen, G. Plotkin, and G. Winskel, "Petri Nets, Event Structures and Domains, Part I", *Theoretical Computer Science*, Vol. 13, pp. 85–108, 1981.

[31] Tobias Nipkow and Konrad Slind, "I/O Automata in Isabelle/HOL", Proc. of TYPES Workshop, 1994. (To appear in Springer-Verlag's LNCS series.)

[32] Amir Pnueli, "The Temporal Logic of Programs", *Proc. of the 18th IEEE Symp. on Foundations of Computer Science*, pp. 46–57, 1977.

[33] Adrian Segall, "Distributed Network Protocols", *IEEE Trans. on Information Theory*, Vol. 29, No. 1, pp. 23–35, Jan. 1983.

[34] M.W. Shields, "Concurrent Machines", *The Computer Journal*, Vol. 28, No. 5, pp. 449–465, 1985.

[35] J. Søgaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants, "Computer-Assisted Simulation Proofs", in [13], p. 305–319.