Mechanical Verification of Concurrent Systems with TLA

Urban Engberg
Aarhus University

Peter Grønning
Technical University of Denmark

Leslie Lamport
Digital Equipment Corporation
Systems Research Center

14 September 1992 – version 2

THIS PAGE SHOULD NOT APPEAR IN THE CAV PROCEEDINGS.

All but this page to appear in *Proceedings of the Fourth International Workshop on Computer-Aided Verification*

Mechanical Verification of Concurrent Systems with TLA

Urban Engberg[1], Peter Grønning[2], and Leslie Lamport[3]

[1] Aarhus University
[2] Technical University of Denmark
[3] Digital Equipment Corporation
Systems Research Center

Abstract. We describe an initial version of a system for mechanically checking the correctness proof of a concurrent system. Input to the system consists of the correctness properties, expressed in TLA (the temporal logic of actions), and their proofs, written in a humanly readable, hierarchically structured form. The system uses a mechanical verifier to check each step of the proof, translating the step's assertion into a theorem in the verifier's logic and its proof into instructions for the verifier. Checking is now done by LP (the Larch Prover), using two different translations—one for action reasoning and one for temporal reasoning. The use of additional mechanical verifiers is planned. Our immediate goal is a practical system for mechanically checking proofs of behavioral properties of a concurrent system; we assume ordinary properties of the data structures used by the system.

1 Introduction

TLA, the Temporal Logic of Actions, is a logic for specifying and reasoning about concurrent systems. Systems and their properties are described by logical formulas; the TLA formula $\Pi \Rightarrow \Phi$ asserts that the system represented by Π satisfies the property, or implements the system, represented by Φ. TLA is a linear-time temporal logic [4] that can express liveness (eventuality) as well as safety (invariance) properties. Although TLA is a formal logic, the TLA specification of a concurrent system is no more difficult to write than the system's description in a conventional programming language.

Since TLA is a formal logic, it allows completely rigorous reasoning. It is clear that TLA proofs can, in principle, be checked mechanically. In 1991, we began a one-year effort to determine if mechanical verification with TLA is practical. We decided to use the LP verification system [1, 2], initially planning to write proofs directly in LP. However, we found the LP encoding to be distracting, making large proofs difficult. We therefore decided to write a TLA to LP translator, so specifications, theorems, and proof steps could be written in TLA. Writing specifications and theorems directly in TLA avoids the errors that can be introduced when hand translating what one wants to prove into the language of a verifier. The translator also allows mechanically checked proofs to have the same structure as hand proofs, making them easier to write and understand.

Working on the translator has thus far allowed us time to verify only a few simple examples, including an algorithm to compute a spanning tree on an arbitrary graph. Experience with more realistic examples is needed to determine if mechanical verification of TLA formulas can be a practical tool for concurrent-system design. It will never be easy to write rigorous proofs. However, we find it very encouraging that mechanically checkable proofs written in the translator seem to be only two to three times longer than careful hand proofs.

TLA, its LP encoding, and the translator are described in the following three sections, using the spanning-tree algorithm as an example. In this example, we prove that a system satisfies a property. An important feature of TLA is the ability to prove that one system implements (is a refinement of) another. However, space does not permit an example of such a proof.

2 TLA

For the purposes of this paper, we can consider TLA to be ordinary predicate logic, except with two classes of variables called *rigid variables* and *flexible variables*, extended with the two operators $'$ (prime) and $\Box$. Quantification is over rigid variables only. (Full TLA also includes quantification over flexible variables, which serves as a hiding operator.) We often refer to flexible variables simply as variables.

The semantics of TLA is based on the concept of a *state*, which is an assignment of values to (flexible) variables. The meaning of a TLA formula is a set of *behaviors*, where a behavior is a sequence of states. The operator $\Box$ is the standard temporal-logic "always" operator [7]; the prime is a "next-state" operator. The complete semantics of TLA can be found elsewhere [6]; here we explain TLA informally through simple examples.

As a first example, we write a TLA formula specifying a program that starts with the variable x equal to any natural number, and keeps incrementing x by 1 forever. Letting *Nat* denote the set of natural numbers, the obvious way to express this with the prime and $\Box$ operators is

$$(x \in Nat) \;\wedge\; \Box(x' = x + 1) \tag{1}$$

The predicate $x \in Nat$ asserts that the value of x in the initial state is an element of *Nat*; the action $x' = x + 1$ asserts that the value of x in the next state is always 1 greater than its value in the current state, and the temporal formula $\Box(x' = x + 1)$ asserts that this is true for all steps—that is, for all pairs of successive states.

Formula (1) asserts that the value of x is incremented in each step of a behavior. For reasons explained in [5], we want also to allow steps that leave x unchanged. Letting $[\mathcal{A}]_f$ denote $\mathcal{A} \vee (f' = f)$, this is expressed by the TLA formula

$$(x \in Nat) \;\wedge\; \Box[x' = x + 1]_x \;\wedge\; \mathrm{WF}_x(x' = x + 1) \tag{2}$$

The conjunct $\Box[x' = x + 1]_x$ asserts that every step of the program either increments x by 1 or leaves it unchanged. It allows behaviors in which x remains unchanged forever. The conjunct $\mathrm{WF}_x(x' = x+1)$ expresses the liveness property that infinitely many $x' = x+1$ steps (ones that do increment x) occur. The reader is referred to [6] for an explanation of the WF operator and its definition in terms of $'$ and $\Box$.

In general, the canonical form of a TLA formula describing an algorithm is

$$Init \ \wedge \ \Box[\mathcal{N}]_v \ \wedge \ F \tag{3}$$

where *Init* is a predicate describing the initial state, $\mathcal{N}$ is an action describing how the variables may change, v is the tuple of all program variables, and F is a liveness condition. The conjunct $\Box[\mathcal{N}]_v$ asserts that every step is either an $\mathcal{N}$ step or else leaves all variables unchanged (since a tuple is unchanged iff every component is unchanged).

Our major example is a simple algorithm that, given a finite connected graph and a root, constructs a spanning tree. For each node n, the algorithm computes the distance $d[n]$ from n to the root and, if n is not the root, its father $f[n]$ in the spanning tree.

When the algorithm is expressed formally, d and f are variables whose values are functions with domain equal to the set of nodes. Before describing the algorithm, we introduce some notation for expressing functions. The expression $\lambda x \in S : e(x)$ denotes a function f whose domain is S, such that $f[x]$ equals $e(x)$ for all x in S. If f is a function, then $f[s := v]$ is the function that is the same as f except with $f[s] = v$. This is defined formally as follows, where *dom f* denotes the domain of f, and $\triangleq$ means *equals by definition*.

$$f[s := v] \ \triangleq \ \lambda x \in dom\, f : \mathbf{if} \ \ x = s \ \ \mathbf{then} \ \ v \ \ \mathbf{else} \ \ f[x]$$

(Thus, $s \notin dom\, f$ implies $f[s := v] = f$.) If f is a function and T a set, then $f[s :\in T]$ is the set of all functions $f[s := v]$ with $v \in T$. Finally, $[S \rightarrow T]$ denotes the set of all functions f with domain S such that $f[x] \in T$ for all $x \in S$.

We now describe the spanning-tree algorithm. Initially, $d[n]$ equals 0 for the root and equals ∞ for all other nodes. For each node n, there is a process that repeatedly executes *improvement steps* that choose a neighbor m with $d[m] + 1 < d[n]$, decrease $d[n]$, and set $f[n]$ to m. The improvement step could simply decrease $d[n]$ to $d[m]+1$, but for reasons that are irrelevant to this discussion, we consider a more general algorithm in which $d[n]$ is set to a nondeterministically chosen number between its old value and $d[m]$. The algorithm terminates when no more improvement steps are possible.

The TLA formula Π describing this algorithm is defined in Figure 1, where *Node* is the set of nodes, *Root* is the root, $Nbrs(n)$ is the set of neighbors of node n in the graph, and $[a, b)$ is the set of natural numbers c such that $a \leq c < b$. We adopt the convention that a list bulleted with $\wedge$'s denotes the conjunction of the items, and we use indentation to eliminate parentheses. We have found this convention extremely helpful in making large formulas easier to read.

The initial condition is described by the predicate *Init*. It asserts that $d[n]$ has the appropriate value (0 or ∞) and that $f[n]$ is a node, for each $n \in Node$.

Action $\mathcal{N}_2(n, m)$ describes an improvement step, in which $d[n]$ is decreased and $f[n]$ set equal to m. However, it does not assert that m is a neighbor of n. The action is enabled only if $d[m] + 1 < d[n]$. (In this formula, d and f are flexible variables, while m and n are rigid variables.)

Action $\mathcal{N}$ is the disjunction of the actions $\mathcal{N}_2(n, m)$ for every node n and neighbor m of n. It is the next-state relation of the algorithm, describing how the variables d and f may change. We define v to be the pair (d, f) of variables, and Π to be the canonical formula describing the algorithm. The weak fairness condition $\mathrm{WF}_v(\mathcal{N})$ asserts that $\mathcal{N}$ steps are eventually taken as long as they remain possible—that is, as long as the action $\mathcal{N}$ remains enabled. Concurrency is represented by the nondeterministic interleaving of the different processes' (atomic) improvement steps.

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ d = \lambda n \in \mathit{Node} : \textbf{if}\ n = \mathit{Root}\ \textbf{then}\ 0\ \textbf{else}\ \infty \\
& & \land\ f \in [\mathit{Node} \rightarrow \mathit{Node}] \\
\mathcal{N}_2(n, m) & \triangleq & \land\ d[m] \neq \infty \\
& & \land\ d' \in d[n :\in [d[m] + 1,\ d[n])] \\
& & \land\ f' = f[n := m] \\
\mathcal{N} & \triangleq & \exists n \in \mathit{Node} : \exists m \in \mathit{Nbrs}(n) : \mathcal{N}_2(n, m) \\
v & \triangleq & (d, f) \\
\Pi & \triangleq & \mathit{Init} \ \land\ \Box[\mathcal{N}]_v \ \land\ \mathrm{WF}_v(\mathcal{N})
\end{array}
$$

Fig. 1. The spanning-tree algorithm.

The correctness property to be proved is that, for every node n, the values of $d[n]$ and $f[n]$ eventually become and remain correct. Letting $\mathit{Dist}(n, m)$ denote the distance in the graph between nodes n and m, the correctness of these values is expressed by the predicate *Done*, defined to equal

$$
\begin{array}{l}
\forall\, n \in \mathit{Node} : \ \land\ d[n] = \mathit{Dist}(\mathit{Root}, n) \\
\qquad\qquad\qquad \land\ 0 < d[n] < \infty \ \Rightarrow\ \land\ f[n] \in \mathit{Nbrs}(n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{Dist}(\mathit{Root}, f[n]) = \mathit{Dist}(\mathit{Root}, n) - 1
\end{array}
$$

(If the graph is not connected, then for every node n not in the root's connected component, *Done* asserts only that $d[n] = \infty$.) The assertion that *Done* eventually becomes and remains true is expressed by the TLA formula $\Diamond\Box \mathit{Done}$, where $\Diamond F$, read *eventually* F, is defined to equal $\neg\Box\neg F$. Correctness of the algorithm is expressed by the formula $\Pi \Rightarrow \Diamond\Box \mathit{Done}$, which asserts that $\Diamond\Box \mathit{Done}$ holds for every behavior satisfying Π.

The usual first step in reasoning about a concurrent algorithm is to prove an invariant. The appropriate invariant *Inv* for our algorithm is the following, where $\setminus$ denotes set difference.

$$\begin{array}{l}
\land\ d \in [Node \rightarrow Nat \cup \{\infty\}] \\
\land\ f \in [Node \rightarrow Node] \\
\land\ d[Root] = 0 \\
\land\ \forall n \in Node \setminus \{Root\} : d[n] < \infty \ \Rightarrow \ \begin{array}[t]{l} \land\ Dist(Root, n) \leq d[n] \\ \land\ f[n] \in Nbrs(n) \\ \land\ d[f[n]] < d[n] \end{array}
\end{array}$$

The invariance of *Inv* is asserted by the formula $\Pi \Rightarrow \Box Inv$. For brevity, we prove the invariance only of the first two conjuncts of *Inv*, which we call TC. A careful hand proof of the TLA formula $\Pi \Rightarrow \Box TC$ expressing the invariance of TC appears in Figure 2 and is discussed below. A similar proof for the complete invariant *Inv* takes about two pages, but has the same basic structure.

The proof in Figure 2 uses a structured format that we find quite helpful for managing the complexity of proofs. Step 1 proves that TC holds in the initial state. Step 2 proves that any single step starting in a state with TC true leaves it true. Step 3 applies the following standard TLA proof rule [6], where I' denotes the formula obtained from I by replacing x with x', for each flexible variable x.

$$\text{INV1}: \quad \frac{I \land [\mathcal{N}]_v \ \Rightarrow \ I'}{I \land \Box[\mathcal{N}]_v \ \Rightarrow \ \Box I}$$

The theorem follows trivially from steps 1 and 3. Step 2, the “induction step”, is the major part of an invariance proof. For this simple invariant, its proof is easy.

The proof that $\Pi \Rightarrow \Box Inv$ is like the invariance proof for TC, except step 2 is more difficult. The proof of the correctness property $\Pi \Rightarrow \Diamond\Box Done$ then uses ordinary temporal-logic reasoning and the TLA proof rule WF1 [6]; space does not permit its description.

3 Encoding TLA in LP

LP is based on a fragment of multisorted first-order logic. To reason about TLA with LP, one must encode TLA formulas in LP's logic. Our initial plan was to have a single encoding. However, as Figure 2 shows, two different kinds of reasoning are used in TLA proofs: steps 1 and 2 illustrate *action reasoning*, not involving temporal operators; step 3 illustrates *temporal reasoning*. Since it is formally a special case, action reasoning is possible in any encoding that allows temporal reasoning. However, such reasoning can be made easier with a special encoding for formulas not containing the temporal operator $\Box$. Action reasoning is almost always the longest and most difficult part of a proof, so we decided to use separate encodings for the action and temporal reasoning. We have found

Theorem $\Pi \Rightarrow \Box TC$

1. $Init \Rightarrow TC$
   Proof We assume $Init$ and prove TC.
   1.1. $d \in [Node \rightarrow Nat \cup \{\infty\}]$
   Proof By definition of $Init$, considering separately the cases $n = Root$ and $n \neq Root$.
   1.2. $f \in [Node \rightarrow Node]$
   Proof By definition of $Init$.
   qed Step 1 follows from 1.1, 1.2, and the definition of TC.
2. $TC \wedge [\mathcal{N}]_v \;\Rightarrow\; TC'$
   Proof We assume TC and $[\mathcal{N}]_v$ and prove TC'.
   2.1. $\mathcal{N} \Rightarrow TC'$
   Proof Since $\mathcal{N} = \exists n \in Node,\ m \in Nbrs(n) : \mathcal{N}_2(n, m)$, it suffices to assume $n \in Node$, $m \in Nbrs(n)$, and $\mathcal{N}_2(n, m)$, and to prove TC'.
   2.1.1. $d' \in [Node \rightarrow Nat \cup \{\infty\}]$
   Proof By definition of TC and $\mathcal{N}_2$, since $[d[m] + 1,\ d[n]) \subseteq Nat \cup \{\infty\}$.
   2.1.2. $f' \in [Node \rightarrow Node]$
   Proof By definition of TC and $\mathcal{N}_2$, since $Nbrs(n) \subseteq Node$, for all nodes n.
   qed Step 2.1 follows from 2.1.1, 2.1.2, and the definition of TC.
   2.2. $(v' = v) \;\Rightarrow\; TC'$
   Proof Follows trivially from the definitions.
   qed Step 2 follows from 2.1 and 2.2, since $[\mathcal{N}]_v = \mathcal{N} \vee (v' = v)$.
3. $TC \wedge \Box[\mathcal{N}]_v \;\Rightarrow\; \Box TC$
   Proof By step 2 and rule INV1.

qed The theorem follows from 1, 3, and the definition of Π.

Fig. 2. The proof of invariance of TC.

the resulting simplification of action reasoning to be worth the inconvenience of having two different encodings.

The encoding of action reasoning in LP is straightforward. TLA's rigid variables become LP variables. For each TLA flexible variable x, we encode x and x' as two distinct LP constants. Thus, the TLA action $(x' = x + 1) \wedge (y' = y)$ is encoded in LP as `(x'=x+1)&(y'=y)`.

The encoding of temporal reasoning is more subtle. In TLA, a formula is an assertion that is true or false for a behavior. Let $\sigma \models F$ denote that the behavior σ satisfies the TLA formula F. Formula F is valid iff $\sigma \models F$ is true for all behaviors σ. The validity of F is represented in LP's logic by $\forall \sigma : \sigma \models F$. Neglecting details of the precise ASCII syntax, this formula is written in LP as $\sigma \models F$, universal quantification over the free variable σ being implicit. The semantic operator $\models$, which cannot appear in a TLA formula, becomes part of the formula's LP translation.

TLA's (temporal) proof rules have straightforward translations into LP. For example, the proof rule INV1 asserts

$$\frac{\forall \sigma : \sigma \models (I \wedge [N]_f \Rightarrow I')}{\forall \sigma : \sigma \models (I \wedge \Box[N]_f \Rightarrow \Box I)}$$

In this rule, $\wedge$, $\Rightarrow$, and $'$ are operators declared in LP to represent the corresponding TLA operators. In particular, $\wedge$ and $\Rightarrow$ are different from LP's built-in conjunction (`&`) and implication (`=>`) operators. Propositional reasoning about temporal formulas is done in LP using such axioms as

$$\sigma \models (F \wedge G) \quad = \quad (\sigma \models F) \ \&\ (\sigma \models G)$$

4 The Translator

The TLA translator is a program written in Standard ML [3] that translates "humanly readable" TLA specifications and proofs into LP proof scripts. Their readability makes proofs easier to maintain when the specifications change than they would be if written directly in LP. The different encodings for action reasoning and temporal reasoning are translated into two separate LP input files. Formulas proved in the action encoding are asserted in the temporal encoding. The proof succeeds if LP successfully processes both files.

4.1 Specifications

Figure 3 is the input to the TLA translator corresponding to the spanning-tree algorithm of Figure 1. (All translator input is shown exactly as typed by the user, except that multiple fonts have been used for clarity.) It begins with a declaration of *Span* as the name of the specification, followed by a directive to read the file *frame*, which contains declarations of all constants such as 0, +, *Nbrs*, and *Node*. The next two lines declare d and f to be (flexible) variables. (In TLA's typeless logic, the only sorts are Boolean and Any.) The rest of the specification is a direct transliteration of Figure 3, except for two differences: the action $N1(n)$ is defined for use in the proofs, and `*` is used instead of comma to denote ordered pairs. The translation of these definitions into LP rewrite rules is straightforward, except for quantified expressions and the lambda-construct. LP does not now support full first-order quantification, so we have defined LP operators and associated proof rules for quantification and lambda abstraction. Each occurrence of a quantifier or "lambda" requires the definition of an auxiliary function, which is named for reference in proofs by a term in brackets `[* ... *]`.

In TLA, prime ($'$) is an operator that can be applied to predicates like *Init* and to state functions like v, where priming an expression replaces all variables by their primed versions. In the LP action encoding, primed and unprimed variables become distinct constants, so the prime operator cannot be expressed. The "bar operator" used in refinement [6, Section 9.3.2] and TLA's *Enabled* operator [6, Section 3.7] are similarly inexpressible. The translator must therefore add to the LP encoding rewrite rules explicitly defining such expressions as $Init'$, $\overline{v}$,

Name *Span*
Use *frame*
Variables
    d, f : **Any**
Predicates
    Init == /\ d = **Lambda** n **in** *Node* :
            If n = *Root* **Then** 0 **Else** *infty* [* dist : **Any** *]
        /\ f **in** [*Node* -> *Node*]
Actions
    $N2(n, m)$ == /\ d[m] ~= *infty*
        /\ d' **in** d[n :**in** *openInter*(d[m] + 1, d[n])]
        /\ f' = f[n := m]
    $N1(n)$ == **Exists** m **in** *Nbrs*(n) : $N2(n, m)$ [* n1(n) *]
    N == **Exists** n **in** *Node* : $N1(n)$ [* n *]
Statefunctions
    v == (d * f)
Formulas
    Pi == *Init* /\ [][N]_v /\ **WF**(v, N)

Fig. 3. The spanning-tree algorithm, in the translator's input language.

and *Enabled* $\mathcal{N}$. Definitions for the primed and barred expressions are generated automatically by the translator. Definitions for the *Enabled* predicates must now be provided by the user; future versions of the translator will generate them as well.

4.2 Proofs

The invariant *TC* of our spanning-tree algorithm is specified in the translator's language as

TC == d **in** [*Node* -> *NatInf*] /\ f **in** [*Node* -> *Node*]

where *NatInf* denotes $Nat \cup \{\infty\}$. The hand proof of invariance of *TC* was based on certain tacit assumptions about *Root*, *Node*, and *Nbrs*. The formal statement of these assumptions is the assertion *Assump*, defined in the translator input to be the conjunction of the following two assertions. (Since the set construct has a bound variable, it requires the same kind of auxiliary function used for quantifiers and "lambda".)

Assump1 == *Root* **in** *Node*
Assump2 == **Forall** n **in** *Node* :
        Nbrs(n) = {m **in** *Node* :
            NbrRel(n, m) [* a22(n) *]} [* a21 *]

where *NbrRel* denotes the neighbor relation on the graph. Further assumptions about *NbrRel* are needed for the complete correctness proof of the algorithm.

Figure 4 contains the translator version of the invariance proof of Figure 2. It has the same structure as the hand proof in Figure 2. Steps are numbered in the more compact fashion $\langle level \rangle step$, with **Step**$\langle 2 \rangle 3$ denoting the third substep of level two and **Hyp**$\langle 1 \rangle.2$ denoting the second hypothesis of level one of the current proof.

The proof is written in a natural deduction style, the translator input **Assume** A **Prove** B denoting that $A \Rightarrow B$ is to be proved by assuming A and proving B. The goal B can be omitted if it is the same as the current goal. (In the temporal encoding, assuming A and proving B means assuming $\sigma \models A$ and proving $\sigma \models B$, for an arbitrary constant σ.) The construct **Reduce by** A **To** B expresses an argument of the form "By A it suffices to prove B." It is converted into LP's style of direct reasoning by rearranging the proof steps. **Normalize**, **Apply**, and **Crit** are LP commands. The applied rules, such as BoxElim1, are defined in LP for reasoning about the translator output. In step $\langle 1 \rangle 3$, INV1 applies the INV1-rule and **Crit**'s the current hypotheses with the resulting fact.

Figure 5 shows the LP input in the action-reasoning file generated from step $\langle 3 \rangle 1$ in the proof of step $\langle 1 \rangle 2$. Additional translator constructs allow arbitrary LP input to be inserted into the output, making the full power of LP available through the translator. (Soundness is maintained if no LP `assert` commands are inserted.) Such direct use of LP was not needed in this proof; our ultimate goal is to make it unnecessary in general.

The predicate TC is just one part of the entire invariant Inv. About three more pages of translator input completes the proof of invariance of Inv. The rest of the correctness proof takes about six more pages. These proofs required additional properties of numbers (elements of $NatInf$), functions, and the distance function $Dist$—including the well-foundedness of the ordering on $[Node \rightarrow Nat \cup \{\infty\}]$ defined by $f \leq g$ iff $f[n] \leq g[n]$ for all $n \in Node$. Properties of the natural numbers (associativity of addition etc.) were expressed directly in LP. Properties of the distance function needed for the proof were asserted in the translator input. Although these properties can be proved from more primitive definitions, we have ignored such conventional verification in order to concentrate on the novel aspects of TLA.

5 Future Directions

It is obviously easier to write a TLA proof in TLA than in an LP encoding of TLA. It was not obvious to us how much easier it would be. Our initial experience indicates that writing a proof with the translator can be an order of magnitude faster than doing the proof directly in LP. Such a speed-up is possible only if the proof can be written in the translator with no direct use of LP. We are planning to enhance the translator to eliminate all direct LP reasoning.

Translating the steps of a proof rather than just the property to be proved permits the use of multiple verification methods. The translator now generates separate LP input for action and temporal reasoning. We plan to generate input to other verification systems as well. Steps that are provable by simple temporal

```
Theorem TC
  Assume []Assump  Prove Pi => []TC

Proof
  <1>1 Assume Assump, Init  Prove TC

    <2>1 d in [Node -> NatInf]
    Reduce by Normalize Hyp<1>.2 with Init,
              Apply ProveFuncSpaceLambda to Red
    To  Assume n in Node  Prove d[n] in NatInf

      <3>1 Case n = Root
      Qed by Normalize Hyp<1>.2 with Init

      <3>2 Case n ~= Root
      Qed by Normalize Hyp<1>.2 with Init

    Qed by Cases

    <2>2 f in [Node -> Node]
    Qed by Normalize Hyp<1>.2 with Init

  Qed by Normalize Goal with TC

  <1>2 Assume Assump, TC, [N]_v  Prove TC'

    <2>1 Case N
    Reduce by Normalize Hyp with N
    To  Assume n in Node /\ N1(n)
    Reduce by Normalize Hyp with N1
    To  Assume m in Nbrs(n) /\ N2(n, m)

      <3>1 d' in [Node -> NatInf]

        <4>1 Assume k in openInter(d[m] + 1, d[n])  Prove k in NatInf
        Qed by Normalize Hyp with UseOpenInterval

      Qed by Normalize Hyp<1>.2 with TC,
             Normalize Hyp<2> with N2,
             Apply ProveFuncSpaceUpdateIn to Hyp<1>.2 Hyp<2> Step<4>1

      <3>2 f' in [Node -> Node]

        <4>1 m in Node
        Qed by Normalize Hyp<1>.1 with Assump Assump2

      Qed by Normalize Hyp<1>.2 with TC,
             Normalize Hyp<2> with N2,
             Apply ProveFuncSpaceUpdateEq to Hyp<1>.2 Hyp<2> Step<4>1

    Qed by Normalize Goal with TC

    <2>2 Case Unchanged(v)
    Qed by Normalize Hyp with v,
           Normalize Hyp<1>.2 with TC,
           Normalize Goal with TC

  Qed by Cases

  <1>3 Assume []Assump, TC, [][N]_v  Prove []TC
  Qed by INV1 on Step<1>2

Qed by Normalize Hyp with Pi,
       Apply BoxElim1 to Hyp,
       Crit Hyp with Step1 Step3
```

Fig. 4. Proof of invariance of TC, in the translator's input language.

```
set name Theorem_Tc1_2_1_1
prove in(d', funcSpace(Node, NatInf))
  set name Theorem_TC_2_1_1_1
  prove (in(v_k_, openInter(((d@v_m_c)+1), (d@v_n_c))) => in(v_k_,NatInf))
    resume by =>
    <> 1 subgoal for proof of =>
      normalize Theorem_TC_2_1_1_1ImpliesHyp with UseOpenInterval
      [] => subgoal
    [] conjecture
  set name Theorem_TC_2_1_1
  normalize Theorem_TC_2ImpliesHyp.2 with TC
  normalize Theorem_TC_2_1ImpliesHyp with N12
  apply ProveFuncSpaceUpdateIn to Theorem_TC_2ImpliesHyp.2 Theorem...
  [] conjecture
```

Fig. 5. Translator output for step $\langle 3\rangle 1$ in the proof of step $\langle 1\rangle 2$ of Figure 4.

reasoning will be verified automatically by a decision procedure for propositional temporal logic. Some steps might be proved with a model checker by enumerating all possibilities. We may also investigate the use of theorem provers other than LP as "back ends" for the translator.

References

1. Stephen J. Garland and John V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes on Computer Science*, pages 137–151. Springer-Verlag, April 1989.
2. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, December 1991.
3. Robert Harper, David MacQueen, and Christopher Wadsworth. Standard ML. Internal Report ECS–LFCS–86–2, Edingburgh University, March 1986.
4. Leslie Lamport. 'Sometime' is sometimes 'not never': A tutorial on the temporal logic of programs. In *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, pages 174–185. ACM SIGACT-SIGPLAN, January 1980.
5. Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.
6. Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.
7. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.