# Proof-checking an audio control protocol with LP

W.O.D. Griffioen

Computer Science/Department of Software Technology

**CS-R9570 1995**

Report CS-R9570
ISSN 0169-118X

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

# Proof-checking an Audio Control Protocol with LP

W.O.D. Griffioen

*CWI*

*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

griffioe@cwi.nl

## Abstract

In this paper we report on the use of the Larch Prover to mechanize the correctness proof of the audio control protocol as presented in [BPV94].

*AMS Subject Classification (1991):* 03B05, 03B15, 68Q42, 68Q22, 68Q60.
*CR Subject Classification (1991):* C.2.2, C.2.5, C.3, D.2.4, F.1.1, F.3.1, F.4.1.
*Keywords & Phrases:* Communication protocols, Larch, Manchester encoding, consumer electronics, protocol verification, proof-checking, real-time systems, timed I/O automata, linear hybrid systems.
*Note:* Research supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125.

## 1. Introduction

The systems that are analysed using formal methods increase in size and complexity and so do the proofs. Because it is not realistic to assume that a proof of 20 pages is flawless, the computer is asked for help to check those proofs. The protocol that is subject of investigation here is a fragment of the Enhanced Easy Link (EEL) protocol. This protocol is used by Philips to communicate control information between the components of an audio set (CD, DCC, amplifier, tuner etc). Though a simplified version of the protocol is verified (the same as in [BPV94]) it is still fairly complicated.

As a vehicle to mechanize the proof we used the Larch Prover (LP). LP is a proof-checker for first-order predicate logic and it is based on rewriting. It has been used for protocol verification in a comparable model MMT [LSGL94]. Here we use the Linear Hybrid Systems (LHS) model of [BPV94], the semantics of systems in this model is defined in terms of timed I/O-automata [LV92, LV93, GSSL93].

We think that both general proof checkers and model checkers are useful for protocol verification. Model checkers require the description to be finite in some sense and sometimes the type of questions the system can answer is restricted. On the other hand the advantage of model checking is of course that the questions are answered without any user interaction. When using general mathematical proof checkers almost no restrictions on the description of the system and/or the type of questions exist. But here much more user interaction is required to finish the proof.

As a correctness criterion we use trace-inclusion between the EEL protocol and a specification. This tells us that the EEL protocol behaves like a message buffer with capacity one

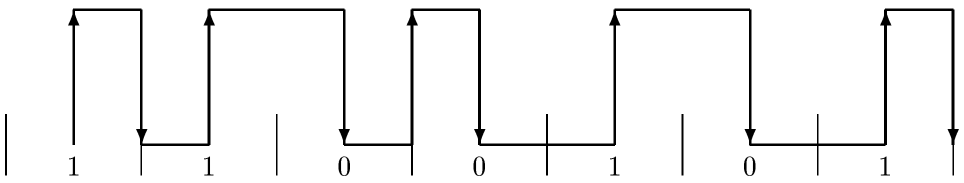

Figure 1: Manchester encoding of 1100101

and that each message is delivered before a specified moment. To prove this we use invariants and a simulation relation.

We formalized the whole proof, including proofs of simple identities like $x * 0 = 0$. Other users of LP [CL92] chose to concentrate on the crucial parts and assume simple properties like

$$((0 \leq X < |Y| * 2^i)\&(Y > 0)) \Rightarrow -|Y| * 2^i \leq X - (Y * 2^i) < 0.$$

We proved facts like these because if they are really simple it should be easy to prove them and when they are not simple it is possible that one makes a mistake. Also, we think it interesting to see what happens if one tries to prove everything from the basic axioms.

This paper is organized as follows. In the next section the EEL protocol is introduced. In Section 3 the LHS model is informally introduced. In Section 4 the model and the protocol are formalized. In Section 5 the Larch Prover is introduced. In Section 6 the correctness proof is formalized. In the last section we discuss the results of our work. In two appendices an example proof is presented and the formal specifications are listed.

## 2. The EEL-protocol

The EEL protocol is used by Philips for communication of control information between the components of an audio set. When for instance only the amplifier has a red-eye to receive the commands of the remote control, the other components receive these commands via the EEL-protocol. It is also used to implement "added intelligence", one can copy a CD to a cassette by pressing a single key: the CD and the cassetedeck are started and when the CD has finished playing the deck stops recording. In fact the components are connected by a small local area network (LAN). This network has one special quality: it is cheap. It uses only one wire, no extra clock wire is needed. Furthermore high tolerances on the timing are allowed, this makes it possible to implement the network via processors that are also used for other tasks.

To transmit messages from one component to the other, Manchester encoding is used. The components are connected by a single wire, the bus. On this wire the voltage can be either high or low. Time is divided in bit-slots of equal length. Bits are transmitted half-way the bit-slots, a bit 1 is transmitted by changing the voltage from low to high, a bit 0 is transmitted by changing the voltage in the other direction. If the same bit is transmitted twice in a row, the voltage changes exactly in between. See Figure 1 for an example.

This is the basic idea but some problems must be solved:

1. The down-going edges are not sharp enough to be accurately detected, so the receiver

only detects the up-going edges. To make correct reception of messages still possible, a restriction is needed: only messages of odd length or ending in 00 are valid.

2. The receiver does not know when the first bitslot of a message starts. Therefore each message starts with a bit-1 and between messages the voltage on the wire is low.

3. The receiver does not know the length of the message it is receiving.

4. All timing is inexact because the protocol has to share one micro processor with several other processes. The Philips documentation of the protocol allows a tolerance of $\pm 5\%$ on all timings.

5. Arbitration is needed when two senders start transmitting at the same time.

6. The delay on the bus can be significant.

Problems 5 and 6 are not addressed in this paper, problem 5 is analysed in [Gri94].

## 3. Linear Hybrid Systems

In this section we give an informal introduction into the LHS model of [BPV94]. The semantics is defined using "old-fashioned recipes" [AL92], in a layered fashion. The I/O automata model is based on *labeled transition systems*. In the untimed case the transition labels can be *input* and *output actions*, which model the interactions of a system with its environment, and *internal actions*, which model internal computation steps. In [LV92, LV93] it is shown how real-time systems can be represented as labeled transition systems by adding, as additional labels, *time-passage actions*. In the resulting model of *timed I/O automata* the continuous progress of real-time is represented by a continuum of discrete time-passage transitions. The LHS model can be viewed as a subclass of timed I/O automata. [BPV94]

The precondition-effect-style is used to define the LHS-systems. Each state of the labeled transition system is described by a valuation of the state variables. When the **init** predicate holds for a state, it is a start state. For every state variable $v$ we have a primed variable $v'$ to denote the new value of $v$ after a transition. The labeled transitions are described by the precondition and effect functions. When the precondition holds in a state $s$ and the effect on $s$ is $s'$ then a transition from $s$ to $s'$ exists.

In Figure 2 an example of a LHS in the style of [BPV94] is given. The system receives a message by a $IN$ action and after $\mathsf{length}(message)$ time units the message is transmitted by an $OUT$ action. The clock ($x$) is not very accurate: when time elapses by $d$ the clock is advanced by an amount between $0.5d$ and $1.5d$. The discrete variables ($list$ in the example) are not allowed to change during a time step, this is expressed by $Unchanged(list)$ in the $TIME(d)$ action predicate. Now we come to a somewhat subtle point. Suppose the length of the message is 5, then the $OUT$ action must occur when $x$ equals 5. What happens if $x$ equals 4, the time action takes a "giant" leap to a new state where $x$ equals 6? That is: the $TIME(d)$ action didn't allow the $OUT$ action to occur? This scenario is not possible because $Stable(Below(x, prec(E, OUT(list))))$ states that if the $OUT(list)$ action is possible now or in the future this should still hold after the time step. The function $prec(E, OUT(list))$ returns

**Discrete**: $list$ : **List**
**Continuous**: $x$ : **Time**

**init:** $list = \epsilon \wedge x = 0$

```
$IN(m)$
    **Effect**
        $list := m;\ x := 0$

$OUT(list)$
    **Precondition**
        $x = \mathsf{length}(list) \wedge list \neq \epsilon$
    **Effect**
        $list := \epsilon$

$TIME(d)$
    **Action formula**
        $\wedge\ d > 0$
        $\wedge\ Unchanged(list)$
        $\wedge\ (0.5 * d) \leq x' - x \leq (1.5 * d)$
        $\wedge\ Stable(Below(x, prec(E, OUT(list))))$
```

Figure 2: Example of LHS-system

the precondition of the action $OUT(list)$ of the system $E$. The other functions used in the action formula of the $TIME(d)$ action are defined as follows.

For $W$ a finite set of unprimed variables, $\phi$ an unprimed formula, and $x$ an unprimed variable:

$$\begin{array}{lcl} Unchanged(W) & \triangleq & \bigwedge_{w \in W} w'=w \\ Stable(\phi) & \triangleq & \phi \rightarrow \phi' \\ Below(x,\phi) & \triangleq & \exists x' : x \leq x' \wedge \phi[x'/x]. \end{array}$$

Note that system E forgets the former message when a new message is received before the current message is transmitted.

In LHS-systems a clock is just a continuous variable, updated by the time action. A clock can be inspected and reset by all actions, compared to other values or other clocks. Furthermore a clock can differ from the ideal clock (see example), so a lot of other things can be described using continuous variables (water-levels, leaked gas). Of course the behaviour of the time action is restricted: for instance two steps of one time-unit should result in the same state as one step of two time-units. For a formal definition of the restriction on the time action we refer to [BPV94].

## 4. Formal specification

To formalize part of the LHS-model and the protocol the Larch Shared Language (LSL) is used. LSL is a first-order algebraic specification language, also used to specify software. Besides LSL the Larch family consists of several other languages, the Larch interface languages. These make it for instance possible to specify a program partly in LSL and partly in programming language such as C and still do type checking.

In this paper LSL is used as a front-end for the Larch Prover (LP). The LP input corresponding to a specification in *filename* can be generated automatically by `lsl -lp` *filename*.

We will start with the LSL specification of the List and Time data-types. Then we will formalize part of the LHS model, followed by a formalization of the EEL protocol. Finally the correctness criterion will be given.

### *4.1 Lists and Time in LSL*

In this section we will give the List and Time traits that we will use in the specification of the EEL protocol. The List data type will be used in the protocol to store the messages. Here we will use the specification as a LSL example. We will present the LSL specification piece by piece.

1

```
List : trait
  includes Bit, Nat
```

In the first line the name of the trait (module) is given. In the second line the traits `Bit` and `Nat` are included. Including a trait is taking the union of the `introduces` and `asserts` clauses of the current trait and the included traits.

2

```
introduces
  head     : List        -> Bit
  tail     : List        -> List
  last     : List        -> Bit
  last_two : List        -> List
  length   : List        -> Nat
  empty    :             -> List
  __ ^ __  : List , List -> List
  [__]     : Bit         -> List
  finalize : List        -> List
```

The `introduces` clause introduces new function symbols and their types.

3

```
asserts
  List generated by empty,[__]:Bit->List,^
  \forall d,e : Bit, m,l,l1 : List
  head(empty)             = 0;
  head([d]^m)             = d;
  tail(empty)             = empty;
  tail([d]^m)             = m;
  last(empty)             = 0;
  last(m ^ [d])           = d;
  last_two(empty)         = empty;
  last_two([d])           = [d];
  last_two(m ^ [d] ^ [e]) = [d] ^ [e];
  length(empty)           = 0;
  length([d] ^ m)         = s(length(m));
  finalize(m)             = (if last(m) = 1 /\ odd(length(m))
                              then m
                              else m ^ [0]);
```

A `generated by` clause asserts that a list of operators is a complete set of generators for a sort. That is, each value of the sort is equal to one that can be written as a finite number of applications of just those operators, and variables of other sorts [GH93].

Lists are generated by the empty list (`empty`), lists with one element (`[__]: Bit -> List`) and the concatenation operator (`^`). The axioms for the functions head, tail, last, last_two, length and finalize are taken from [BPV94]. The finalize function is specific for the specification of the EEL protocol. It is used in the specification of the receiver. When necessary it adds a 0-bit to a message.

4

```
  ((l ^ l1) ^ m)          = (l ^ (l1 ^ m));
```

```
    m ^ empty                 = m ;
    empty ^ m                 = m ;

   [d] ^ l ~= empty ;
    d = e /\ l = l1 <=> [d] ^ l = [e] ^ l1
```

---

These axioms complete the specification of our List data-type. These were implicitly assumed in the handwritten proof, but we had to make them explicit in the formalization.

Note that the booleans and the conjunction (/\) are part of the LSL language. Also the **if then else** construct is part of LSL.

Each well-formed trait defines a theory in a multisorted first-order logic with equality. Each theory contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. This loose semantic interpretation guarantees that formulas in the theory follow only from the presence of assertions in the trait - never from their absence. Using the loose interpretation ensures that all theorems proved about an incomplete specification remain valid when it is extended. (page 37 [GH93]).

In Appendix B.6 the `Time` trait is given. The continuous variables of LHS are of type Real. As mentioned in [BPV94], for the purpose of this verification any interpretation of Real as an ordered field will do: the only properties of reals that we use are the axioms for ordered fields. So the `Time` trait contains essentially the properties of an ordered field. For notational convenience some functions and constants are added, like the integer numbers 2 .. 20, the other inequalities ($\geq, <, >$) and the min function. Specific for this verification a constant T, which denotes the drifting of the clocks, is added. Furthermore an "almost" equal operator $\approx$ is defined that will be used in the proof. In LSL and LP there is no need for functions to be total so we did not add $0^{-1} = 1$ as in [BPV94].

### *4.2 The LHS model*

In this section the LSL-specification of the LHS-model is given. That is, a part of the model: traces, executions and simulations. Not defined in this paper are: composition of systems, hiding and liveness. As starting point we used traits for MMT-automata that are presented in [LSGL94]. The main difference is the way time is handled, the adaption comes down to deleting some traits (the ones that handle time intervals) and slightly adapting the others.

In Figure 3 the trait `System` is depicted. This trait contains the basics for all systems, every trait that defines a LHS-system will include this one. In this trait are the declarations for the underlying labeled transition system: the sort `States[A]` corresponding to the states of the system, the function `start` denoting the set of start states, the function `isStep` denoting the transition-relation. Using the brackets [] finite sequences are constructed, $s_1 a_1 s_2 a_2 s_3$ is written as $[s_1][a_1, s_2][a_2, s_3]$. The function `execFrag` tests if a sequence is an execution-fragment and the function `trace` returns the list of visible actions from a sequence. The `common` function is needed because in the formalization every system has its own sort for actions and sorts are disjunct in LSL. The `common` function maps the actions that systems have in common to a new sort `CommonActions` (see appendix B.3). This makes it possible to compare the actions and traces of different systems.

```
System(A):trait
 introduces
   start      : States[A]                          -> Bool
   pre        : States[A], Actions[A]              -> Bool
   eff        : States[A], Actions[A], States[A]   -> Bool
   isStep     : States[A], Actions[A], States[A]   -> Bool
   [__]       : States[A]                          -> StepSeq[A]
   __[__,__]  : StepSeq[A], Actions[A], States[A]  -> StepSeq[A]
   execFrag   : StepSeq[A]                         -> Bool
   first, last : StepSeq[A]                        -> States[A]
   isVisible  : Actions[A]                         -> Bool
   common     : Actions[A]                         -> CommonActions
   empty      :                                    -> Traces
   __ ^ __    : Traces, CommonActions              -> Traces
   trace      : Actions[A]                         -> Traces
   trace      : StepSeq[A]                         -> Traces
   reachable  : States[A]                          -> Bool
 asserts
   StepSeq[A] generated by [__], __[__,__]
   Traces generated by empty, ^
   \forall a,a' : Actions[A], s,s' : States[A], ss: StepSeq[A]
     isStep(s, a, s') <=> pre(s,a) /\ eff(s, a, s');
     execFrag([s]);
     execFrag(ss[a,s]) <=> execFrag(ss) /\ isStep(last(ss),a,s);
     first([s]) = s;  first(ss[a,s]) = first(ss);
     last([s]) = s;  last(ss[a,s]) = s;
     trace(a) = (if isVisible(a) then empty ^ common(a) else empty);
     trace([s]) = empty;
     trace(ss[a,s]) = (if isVisible(a) then trace(ss) ^ common(a) else trace(ss));
     reachable(s) <=> \E ss (execFrag(ss) /\ start(first(ss)) /\ last(ss) = s)
```

Figure 3: LSL trait System

```
Forward(A,B,f) : trait
  assumes System(A), System(B)
  introduces f : States[A], States[B] -> Bool
  asserts \forall s, s' : States[A], u : States[B], a : Actions[A],
         alpha : StepSeq[B]
  start(s) => \E u (start(u) /\ f(s,u));
  f(s,u) /\ isStep(s,a,s') /\ reachable(s) /\ reachable(u) =>
    \E alpha (execFrag(alpha) /\ first(alpha) = u /\
              f(s', last(alpha)) /\ trace(alpha) = trace(a))
```

Figure 4: LSL trait Forward.lsl

We say that a system implements another system when the set of traces of A is a subset of the set of traces of B. To prove this we use a forward simulation relation between the implementation and the specification. In a forward simulation each start state of the implementation is related to at least one start state of the specification. When two states are related and the implementation can do an action a, the specification can also do an action a and the new states are also related. When it has been proved that a forward simulation exists, a meta-theorem (see for instance [LV95]) gives us the trace inclusion. In Figure 4 the notion of a forward simulation is defined. Because it is restricted to the reachable states this is essentially the 'weak forward simulation' of [LV95].

In this paper we use the untimed interpretation of timed systems. In [BPV94] there are input-, output- and internal actions and a special time action. Here we have visible and invisible actions, where the input, output and time actions are visible and the internal actions are invisible. In timed traces each action has a time stamp and the time action itself does not occur as an action in the traces. An untimed trace of timed system is just a sequence of actions and the time action occurs in it like the other visible actions. The untimed interpretation is sound in the sense that trace inclusion in the untimed interpretation implies timed trace inclusion in the timed interpretation. We chose to work with this untimed interpretation because it is slightly easier to work with. We refer to [LV93] for a formal description of the relation between the timed and the untimed interpretation.

## *4.3 The EEL protocol*

In appendix B.1 the machine readable definition of the protocol is presented. In this section we will point out the differences between the original specification of [BPV94] and the Larch trait `S` in the appendix.

The system defined in trait `S` corresponds to the composition of the sender S and the receiver R with the UP action hidden as internal action (HIDE UP IN (S || R)) of [BPV94]. In this paper the whole system is presented directly instead of presenting the subsystems S and R and the whole system as a composition of these, because the composition operator (||) is not defined in LP.

Some differences between the original-specification [BPV94] and the one given in this paper are caused by abbreviations in the original specification. For assignments the $x := c$ notation is used instead of `x' = c`. The phrase if $b$ then $x := c$ else $x := x$ is abbreviated to if $b$ then $x := c$. When a variable is not assigned a new value, it is assumed to have the same value in the new state, so $x := x$ is added implicitly. These notations are not formalized in LP.

A more substantial difference between the original and the LSL version of the specification is caused by the fact that LP does not support higher-order logic. In the original specification the phrase $\mathit{Stable}(\mathit{Below}(x, \mathit{prec}(S, UP)))$ expresses that: When the precondition of $UP$ holds or can hold in the future the $TIME(d)$ action is not allowed to bring the system in a state where this does not hold. The higher-order functions are replaced by first-order functions that exhibit exactly the same behaviour. For this purpose the function $aux$ is introduced, it is essentially the same as the precondition. The only difference is that the clock variable `x` is added as a parameter. Using this help-function the phrase can be translated to:

```
((\E y (y >= s.S.x /\ aux(s,y,UP))) => (\E z (z >= s'.S.x /\ aux(s',z,UP))))
```

This section is finished by a small piece of a specification in both styles to give an idea of the distance between the two. First the "normal" notation is given followed by the Larch Shared Language version. Note that this is not part of the EEL specification as used in the verification because the error variable is not taken into account.

$IN(m)$
**Precondition**
$\wedge$ head$(m)$=1
$\wedge$ (odd(length$(m)$) $\vee$ last_two$(m)$=$\langle 00\rangle$)
**Effect**
if $\neg transmitting \wedge \neg wire\_high \wedge list{=}\epsilon$ then $[list := m$
$x := 0]$

1
```
%% *** IN(m) ***
pre(s, IN(m)) =
     (head(m) = 1 /\ (odd(length(m)) \/ last_two(m) = [0] ^ [0]));
eff(s, IN(m), s') = (
     (if ~s.S.transmitting /\ ~s.S.wire_high /\ s.S.list = empty
         then s'.S.list = m /\ s'.S.x = 0
         else s'.S.list = s.S.list /\ s'.S.x = s.S.x)
  /\ s'.S.transmitting = s.S.transmitting
  /\ s'.S.wire_high = s.S.wire_high
  /\ s'.R = s.R) ;
```

## *4.4 Correctness criterion*

Beside a specification of the protocol a specification of the desired behaviour is needed, the LSL-definition of this system is presented in appendix B.2.

The specification of system P is slightly different from the original specification [BPV94] which contained a small mistake. The precondition of the $OUT$ action was:

$OUT(list)$
**Precondition**
$\vee(list \neq \epsilon \wedge (1-T)x \leq (4length(list)+5)Q)$
$\vee chaos$

but it should have been:

$OUT(m)$
**Precondition**
$\vee(list = m \wedge list \neq \epsilon \wedge (1-T)x \leq (4length(list)+5)Q)$
$\vee chaos$

This makes a difference when $chaos$ is true, in the former case only $OUT(list)$ actions are allowed while in the new case $OUT(m)$ actions for arbitrary messages $m$ are allowed. Although the first version was on paper, the handwritten proof assumed the second specification.

## 5. Introduction to LP

The Larch Prover (LP) [GG, GH93] is an interactive proof support system. It does not use complicated heuristics to search for a proof. It supports first-order logic and is based on rewriting. When LP is asked to prove a conjecture, it typically normalizes the conjecture using the rewrite-rule versions of the axioms and the lemmas that have already been proved. When a normal-form is reached the proof is suspended and the user can invoke a command. We will mention a few typical options: The user can start a proof by cases, making LP to generate a subgoal for each case. A proof by induction is possible when a sort has a set of generators. This set of generators must be given by the user. LP will then generate a subgoal for each generator. An other possibility is to apply a rewrite rule in the reversed direction, this is allowed because the rewrite-rules are oriented axioms, not implications. When quantifiers are involved, variables or constants can be fixed, specialized or generalized. Furthermore LP can compute critical-pairs and complete a set of rewrite-rules. Besides these proof-commands LP has commands to direct the orientation of axioms into rewrite-rules, to make rewrite-rules inactive, to make proof scripts, etc. Because a proof in LP is based on rewriting, the tool is good at it: it is fast and rewriting modulo associativity and commutativity is supported.

As mentioned before the `lsl` tool compiles the LSL-traits into LP scripts. These scripts can be executed from LP to add the axioms of the traits to the current set of facts. The first step of constructing a proof in LP is to orient the axioms into rewrite rules. Several methods are provided to do this. The standard method is a registered ordering based on a LP-suggested partial ordering of operators. This usually works very well without any user interaction. This method is also used to orient the new facts, like assumptions and proved conjectures, that are generated during the proof. We sometimes guided it toward the intended result by providing a small part of the ordering on the function symbols. Another method is polynomial ordering, we did not use this for the proof of this paper. The least elegant are the "brute-force" ordering procedures, which give users complete control over whether equations are oriented from left to right or in the other direction. This method is used sometimes in a proof when we want to use a rewrite rule in the other direction, typically to expand a definition (`2 -> 1 + 1`).

If a set of rules is non-terminating and LP is (apparently) in a rewrite-loop LP will stop the normalization after a number of rewrite steps. This number can be chosen by the user. A reasonable number is one thousand, which is the default value. In our experience this happens hardly ever. If it happens the user can do several things. One can increase the maximum number of rewrite steps and resume the normalization. An offending rule can be made *inactive*, thereafter LP will not use it in rewriting. Another option is to orient a rule in the other direction. A more elegant solution is to use polynomial ordering or a registered ordering to guarantee that the rewrite system is terminating.

Note that LP does not require the user to prove that the set of rules is consistent or terminating. The authors expect that an inconsistency will reveal itself when one starts using the rules in a proof.

When the user is satisfied with the set of rewrite rules the proving can really start. When a proof is non-trivial (and sometimes even when it is trivial) it is necessary to have a fairly detailed handwritten proof before LP is started. It is possible to play around in LP and just try a proof by induction and see what happens. And when it does not work use the `cancel` command to backtrack to the point just before the induction proof was started. In our experience this only worked in very rare cases.

We constructed most of our proofs in LP in several rounds. In the first round we constructed a rough proof. When a subgoal was not interesting but still complicated to prove we just added the goal as an axiom to the system thereby skipping that part of the proof. Although a proof with holes is not a proof at all, it still provides useful information. The user can go to the problematic parts of the proof very fast and begin proving those parts. Hereby he gets a higher confidence in the correctness of the conjecture before starting with the time consuming and boring parts of the proof. Furthermore it sometimes turned out that several ad-hoc axioms were (almost) the same so it was useful to construct a lemma and use it to prove these subgoals, instead of proving the same thing several times. In the next rounds we proved some skipped subgoals, sometimes these proofs also contained some skipped subgoals. We continued in this way till the proof was complete.

This method is possible and reasonable because LP can generate script-files of the commands the user types. The script files are plain ASCII files with neatly indented commands with some extra annotation. When a subgoal is generated LP adds a diamond (`<>`) and when a (sub)goal is proved a box (`[]`) is added. This annotation is useful when the conjecture or the set of axioms is slightly changed and a proof is rerun. When LP runs a script and encounters a box but has not proved a (sub)goal it stops and notifies the user about the problem. Without this annotation LP would execute all following commands to that subgoal and it would be very hard to find the place were the problems started.

To finish the introduction in LP, in the diagrams below two (very) simple LP proofs are presented. The first lemma states that `length(m ^ [d]) = s(length(m))`, while the corresponding axiom is `length([d] ^ m) = s(length(m))`. For this purpose a more general theorem is proved: `length(l ^ l1) = length(l) + length(l1)` by induction on l. For the `l = empty` case LP proves it without user interaction. Note that `x + 0 -> x`, + is commutative and `empty ^ m -> m`. For the case `l = [b4]`, where `b4` is fresh variable of type **Bit**, some user interaction is necessary.

The subgoal now reads: `s(length(l1)) = length([b4]) + length(l1)`. So we have to convince LP that `length([b4]) = s(0)`. We use the `mempty` fact: `m ^ empty -> m` and of the List trait `length([d] ^ m) = s(length(m))`. A critical pair of these is `length([d]) = s(length(empty))`. After this fact has been added, the proof is completed by normalization.

1

```
set name p
prove :list9: length(m ^ [d]) = s(length(m))
  prove length(l ^ l1) = length(l) + length(l1)
    res by ind on l
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      cri-pair List with mempty
      [] basis subgoal
      <> induction subgoal
      [] induction subgoal
    [] conjecture
  [] conjecture
```

---

For the second example we do not give any intuition but only the facts mentioned in the proof.

```
x <= y => (x + z) <= (y + z);              %% TimeF1
x <= y /\ 0 <= z => (x * z) <= (y * z);    %% TimeF2
x <= y \/ y <= x;                          %% TimeL4
```

---

2

```
set name p
prove :time4: (x * x) >= 0
  prove x <= 0 => m(x) >= 0
    ins y by 0, z by m(x) in TimeF1
    [] conjecture
  res by case x >= 0, m(x) >= 0
    <> case justification
    res by case 0 <= x
      <> case 0 <= xc
      [] case 0 <= xc
      <> case ~(0 <= xc)
      ins x by 0, y by xc in TimeL4
      ins x by xc in p
      [] case ~(0 <= xc)
    [] case justification
    <> case xc >= 0
    ins x by 0, y by xc, z by xc in TimeF2
    [] case xc >= 0
    <> case m(xc) >= 0
    ins x by 0, y by m(xc), z by m(xc) in TimeF2
    [] case m(xc) >= 0
  [] conjecture
qed
```

---

## 6. Formalization of the proof

In this section we will report on the proof that there exists a weak forward simulation from the implementation (`S`) to the specification (`P`).

Unlike the handwritten proof the formal proof starts with proving data-identities. Apart from three trivial lemmas over the **Bit** and **Nat** sorts we have a dozen identities over the sort **List**. For instance `last(l) = 1 => ~(last_two(l ^ [0]) = [0] ^ [0])` and `length(m ^ [d]) = s(length(m))`. These are fairly easy to prove with LP. The twelve also include some trivial ones like `last([d]) = d` and `tail([d]) = empty`. These hardly deserve it to be a lemma, the proof consists of applying one rewrite rule in the reversed direction:

`[d] = empty ^ [d] = [d] ^ empty`. But once these identities have been proven, `last([1])` will be rewritten to `1` without further user interaction. This seemed very useful because sometimes a conjecture did not normalize as expected because we assumed `last([1]) = 1` while LP did not know this. And when the logical system contains about four hundred facts, some of them one screen full, it can be hard to find that the reasoning is stuck at `last([1]) = 1`.

We have about thirty lemmas concerning the data type **Time**. Again we have some trivial ones like: `m(0) = 0` where `m(x)` is the negation of `x`[1]. But we also have some lemmas that needed some thought how to prove them in LP. We started with some basic properties like `0 < 1`. For this one we needed about twenty proof commands, we think that this is not an optimal proof: probably both at the abstract mathematical level and at the level of LP commands it is possible to optimize it.

In the proof the relation between the clock of the sender and the receiver is very important. In [BPV94] an operator $\approx$ is used to express that two clocks are approximately the same, that is, equal modulo drifting. It is defined as follows (our ASCII notation for $\approx$ is `#`):

$$x \approx y \triangleq \frac{1-T}{1+T}x \leq y \leq \frac{1+T}{1-T}x$$

About ten of the lemmas contain the $\approx$ operator, these are relatively intricate to prove in LP. An example is: `y <= 8 /\ x # y => x < (y + 1)`. For this one we used about forty proof commands in LP. It is listed in Appendix A. First we constructed a very detailed handwritten proof of ten steps. The LP proof comes down to instantiating facts and explicitly applying rewrite rules in the reversed direction. To prevent that we lose track we constructed a sub proof for every step of the handwritten proof. Of course it is possible to do it without a division in sub proofs but then the proof would consist of a long list of instantiate and rewrite commands and it would not be clear how it corresponds to the handwritten proof. Furthermore the logical system (the set of facts) would get messy. It would contain a lot of instantiated rules and some other rules are made inactive to use them for reversed rewriting. Sometimes such a messy system has unexpected rewrite properties. By using a proof for each step, a proof context is created for each step. These are deleted when the step is proved and only the sub-conjecture is added to the top context.

After the data lemmas the "real" proof starts. The rest of the proof presented here is the Larch formalization of the proof presented in [BPV94], so all definitions are taken from that paper. When there is a difference we will say so.

First we prove some invariants about the state space of the sender. We start with an easy one, it reflects the observation that the sender is always transmitting if the voltage on the bus is high. In LSL invariants are functions with this signature `inv: States[A] -> Bool`.

```
inv(s) = (s.S.wire_high => s.S.transmitting)
```

[1]We do not use the more natural notation $-x$ because in an old release LP could not parse its own output when the unary - was involved.

For every invariant we prove (a) that it holds in the start states and (b) that if a state is reachable and the invariant holds and the system can do an action to a new state the invariant holds also in this new state. In LP this is expressed as follows:

```
    prove
a)    (start(s:States[S]) => inv(s)) /\
b)    (reachable(s:States[S]) /\ inv(s) /\ isStep(s:States[S],a,s') => inv(s'))
    ..
```

Given a and b it follows that (c) holds.

c) `reachable(s) => inv(s)`

In higher order logic we can prove that the implication $a \wedge b \rightarrow c$ holds where `inv` is a variable of type `States[S] -> Bool`. In LP we have a proof that is replicated for each invariant were inv is substituted by the current concrete invariant. This is one of the few cases were the fact that LP is a first order tool is really a disadvantage.

The next invariant gives an upper bound of the clock in the various stages of progress of the sender:

```
invS(s) = (
    (~s.S.transmitting /\ ~s.S.wire_high /\ s.S.list = empty)
 \/ (~s.S.wire_high /\ s.S.list ~= empty /\ ~s.S.transmitting /\ s.S.x = 0)
 \/ (~s.S.wire_high /\ s.S.list ~= empty /\ s.S.transmitting
     /\ head(s.S.list) = 1 /\ s.S.x <= 4)
 \/ (~s.S.wire_high /\ s.S.list ~= empty /\ s.S.transmitting
     /\ head(s.S.list) = 0 /\ s.S.x <= 2)
 \/ (s.S.wire_high /\ s.S.list ~= empty /\ head(s.S.list)=0 /\ s.S.x <= 4)
 \/ (s.S.wire_high /\ (s.S.list = empty \/ head(s.S.list) = 1) /\ s.S.x <= 2))
```

Now we give invariants for relations between the states of the sender and the receiver. The next invariant tells us that during normal operation (`s.error` is false) an input of a new message can only happen when the receiver is at rest. This invariant is slightly different from the one given in [BPV94] where the `s.error` disjunct has been omitted in the conclusion of the implication. In Section 7.1 we discuss this mistake.

```
invFirstBit(s) = ((~s.S.wire_high /\ s.S.list ~= empty /\ ~s.S.transmitting)
                  => s.R.list = empty \/ s.error)
```

For the correctness of the implementation it is very important how the clocks of the sender and the receiver are related. The first invariant gives the possible distances and the second gives a more detailed description. The second differs from the one presented in [BPV94] in the same way as `invFirstBit` differs from the original.

```
invW(s) = (
((s.S.transmitting /\ ~s.S.wire_high =>
          (s.R.x # (s.S.x + 4))
       \/ (s.R.x # (s.S.x + 2))
       \/ (s.R.x # s.S.x /\ head(s.S.list) = 1 ))
  /\ (s.S.transmitting /\ s.S.wire_high   =>
          (s.R.x # s.S.x)
       \/ (s.R.x # (s.S.x - 2) /\ s.S.list ~= empty /\ head(s.S.list) = 0))))
```

```
invX(s) = (
 ((~s.error /\ s.S.transmitting /\ s.R.list ~= empty    ) =>
    (  (last(s.R.list) = 1 /\ s.R.x <= (pTmT * 4)
        /\ s.R.x # s.S.x)
    \/ (last(s.R.list) = 1 /\ (mTpT * 4) <= s.R.x /\ s.R.x <= (pTmT * 8)
        /\ s.R.x # (s.S.x + 4))
    \/ (last(s.R.list) = 0 /\ s.R.x <= (pTmT * 2)
        /\ s.R.x # (s.S.x - 2))
    \/ (last(s.R.list) = 0 /\ (mTpT * 2) <= s.R.x /\ s.R.x <= (pTmT * 6)
        /\ s.R.x # (s.S.x + 2))))
/\ ((  ~s.error /\ ~s.S.transmitting /\ ~s.S.wire_high
   /\ s.S.list = empty /\ s.R.list ~= empty             ) =>
    (  (last(s.R.list) = 1 /\ s.R.x <= 9 /\ s.R.x # s.S.x)
    \/ (last(s.R.list) = 1 /\ (mTpT * 4) <= s.R.x /\ s.R.x <= 9
        /\ s.R.x # (s.S.x + 4))
    \/ (last(s.R.list) = 0 /\ (mTpT * 2) <= s.R.x /\ s.R.x <= 7
        /\ s.R.x # (s.S.x +2)) ))
/\ ((~s.error /\ s.R.list = empty                      ) =>
    (~s.S.transmitting /\ ~s.S.wire_high)))
```

The next invariant implies that during normal operation output of a message by the receiver cannot happen when the sender is still busy.

```
invLO(s) = (
      s.S.list ~= empty
   /\    ((s.R.list ~= empty /\ last(s.R.list) = 0 /\ s.R.x = 7)
      \/ (last(s.R.list) = 1 /\ s.R.x = 9))
=> s.error )
```

The next invariant gives an obvious property of the specification P.

```
invp(u:States[P]) = (
    u.list = empty \/
     (head(u.list) = 1 /\
       (odd(length(u.list)) \/ last_two(u.list) = ([0] ^ [0])) ) )
```

The two simple invariants below are not mentioned in [BPV94].

```
invRX0(s) = (s.R.list ~= empty => s.R.x >= 0)
invSX0(s) = ((s.S.transmitting \/ s.S.list ~= empty) => s.S.x >= 0)
```

Finally we define a simulation relation SIM, defined in LSL as follows:

```
                                                                           1
SIM : trait
  includes S, P
  introduces SIM : States[S], States[P] -> Bool
  asserts \forall s : States[S], p : States[P]
  SIM(s,p) =
  (if s.error
   then p.chaos
   else (if s.R.list = empty
         then p.list = s.S.list /\ (s.S.list = empty \/ p.x = 0)
         else (if s.R.x # (s.S.x + (2 * (t(last(s.R.list)) - 1)))
               then    (p.list = s.R.list ^ s.S.list)
                    /\ ((1-T)*p.x)
                          <=
                       (((4*t(length(s.R.list))) - (2*(1+t(last(s.R.list)))))
                                      +
                         min(s.R.x, s.S.x + (2 * (t(last(s.R.list)) - 1))))
               else    (p.list = s.R.list ^ [0] ^ s.S.list)
                    /\ ((1-T)*p.x)
                           <=
                       (((4*t(length(s.R.list))) - (2*(1+t(last(s.R.list)))))
                                      +
                       min(s.R.x, s.S.x + (2 * (t(last(s.R.list)) + 1)))))))

implies
  Forward(S,P,SIM)
```

Note that a lot of brackets are needed because in Larch it is impossible to define a precedence for self defined operators.

By the `implies` clause at the end of this trait it is claimed that SIM is a forward simulation from S to P. If that can be proved (and we did) then a meta result (see for instance [LV95]) tells us that S is indeed an implementation of P.

How to read the simulation above? The basic idea behind this simulation is that the concatenation of the lists in the implementation (`s.R.list` and `s.S.list`) is about the same as the message in transit (`p.list`). Formally, when the system is transmitting then the following holds:
`(p.list = s.R.list ^ s.S.list) \/`
`(p.list = s.R.list ^ [0] ^ s.S.list).`
This is expressed in the third if-then-else, in the second if-then-else it is tested if the system is (almost) transmitting and in the first it is tested if the system is in an error situation. The big inequations ( `(1-T)*p.x <= ...`) express that the system returns the messages in time.

## 7. Discussion

In this section we will give some conclusions about this case-study. As said in the introduction it is unlikely that a handwritten proof is flawless. In the first subsection we will report on the errors we encountered. In the second subsection we will draw some conclusions about LP and in the last subsection we will compare the approach of this paper with other approaches.

## *7.1 Errors in handwritten proof?*

As mentioned at the end of Section 4.4 we found a small error in the specification of the OUT-action of system P. In some of the invariants a similar error existed, namely that $INV$ was claimed while only $\neg$`error` $\rightarrow INV$ holds.

The source of these mistakes was that the model in which the protocol was described had been changed during the construction of the proof. In the former model states where `error` held were not accepted and traces that ended in a not accepted state were left out of consideration. Parts of the proof in the former model were still valid in the model presented in [BPV94], but unfortunately the subtlety with the `error` variable was overlooked.

From a practical point of view these bugs were not so important because it was very easy to fix them. It was far more time consuming to fix an illegal proofstep in the TIME case of the simulation. There the following implication was assumed: `s,u |= not(X # Y)` $\rightarrow$ `s',u' |= not(X # Y)`. In general this does not hold. For example when `T` equals 0.05 and `s,u |= X = 1 /\ Y = 1.2` and `s',t' |= X = 2 /\ Y = 2.2` then the implication does not hold. We had to make extra case distinctions and do some extra arithmetic.

Because of the distance between the LP-proof and the handwritten proof it is possible that there are other illegal proof steps in the handwritten proof that were not noticed. When a conjecture is successfully proved by LP along similar lines, we did not investigate the handwritten proof.

In the handwritten proof data identities like: $\mathsf{last}(l) = 1 \rightarrow \mathsf{last\_two}(l\hat{}[0]) \neq [0]\hat{}[0]$ are used without a proof. To our taste this is reasonable in a handwritten proof. But when a proof is mechanized in LP, it is necessary to prove such data identities.

For a lot of facts concerning the time domain the same can be said. Facts like $0 < 1$ and $x > 0 \rightarrow (1/x) > 0$ are fairly obvious for humans. In this paper we even proved these simple ones. Less obvious is for instance: $y \leq 8 \wedge x \approx y \rightarrow x < (y + 1)$ . This is not proved in the handwritten proof, to our taste this is an omission. It took some time to formulate them in a proper way and prove them in LP.

## *7.2 About LP*

In this section we will report on the use of LP.

*Installing and starting.* To start at the beginning: it is easy to install LP. Release 3.1a comes with online documentation in HTML files that can be viewed using a WWW viewer. For release 2.4 a paper document [GG91] is available, to start with this is probably easier to read. In about ninety pages the ideas and commands of LP are explained and one can start using LP. The most important difference between the two versions are: Full first-order logic is supported, not just the universal-existential subset supported by Release 2.4. Furthermore a simple sort system for describing polymorphic abstractions is added.

*Statistics.* In Figure 5 the number of occurrences of LP commands is listed. In the LP proofs we also used a number of display commands but these are not included in the list because

| command | # | meaning |
|---|---|---|
| prove | 298 | ask LP to prove a conjecture |
| res by `=>` | 115 | resume the prove by assuming the lhs |
| res by `<=>` | 10 | resume by two cases: `=>` and `<=` |
| res by case | 152 | resume by a case distinction |
| res by spec | 23 | resume by specialization |
| res by contra | 37 | resume by assuming the contradiction of the current goal |
| res by ind | 31 | resume by a proof by induction |
| rew | 92 | rewrite (mostly in reversed direction) |
| ins | 608 | instantiate variables in a fact |
| cri-pair | 4 | compute critical pairs |
| fix | 31 | fix a variable |
| reg | 3 | give part of ordering on function symbols |
| set | 164 | set system variables of LP |
| make | 121 | make facts immune, passive etc. |
| del | 46 | delete facts |
| dec | 46 | declare variables or functions. |

Figure 5: Number of uses of LP commands

| class | # commands | in % |
|---|---|---|
| bit | 6 | 0 |
| naturals | 16 | 1 |
| list | 123 | 7 |
| time | 666 | 37 |
| invariants | 471 | 26 |
| main theorem | 499 | 28 |
| TOTALS | 1781 | |

Figure 6: Number of LP commands used in the proof.

these commands do not influence the proof and could be deleted without harming the proof. Of course the commands often have arguments, but mostly the complete command fits on one line. The total proof script consists of about 4000 lines (150 Kb). Beside the commands it contains lines with annotations, the boxes (`[]`) and the diamonds (`<>`).

The proofs can be divided in different classes: the lemmas over the datatypes, the invariants and the simulation. To gain some insight in the relative complexity of each class, the number of proof commands used to proof all lemmas in a class are depicted in Figure 6.

The proofs concerning time take up more than one third of the total proof. This is caused by the intrinsic complexity of the timing in the protocol, and by the absence of arithmetic procedures in LP. Although the time lemmas are simple arithmetic, our experience is that it is equally hard to find the right lemmas, and in LP the time lemmas are even harder to prove than the invariants.

LP is fast enough to be used really interactive. Running the complete proof script, which contains the proofs for the data-identities, the invariants and the simulation takes about 1:30 hours on a Sun Sparc 10. The total number of proof commands is 1800. So on the average the execution of one command takes 3 seconds.

*Proving* The LP proofs follow the lines of reasoning of the handwritten proofs, that is: the induction schemes and case distinctions are the same. When the handwritten proof refers to an invariant in the LP proof it is mostly sufficient to instantiate the invariant with the current state. The normalization does the rest of the reasoning. But especially when arithmetic is involved the LP proof contains much more details than the handwritten proof.

It is interesting to know how much effort it took to formalize the proof in LP. But in this case it is not possible to give an exact answer. First of all, this was our first project with LP so it took some time to get used to the system. Furthermore the handwritten proof was not error free so we also had to pay attention to the abstract content of the proof. Also in the formal proof we proved the data-identities not present in the handwritten proof. Finally we had to cope with some problems in LP. But we estimate that given a complete and correct handwritten proof it still would take weeks to formalize it in LP.

As said LP is really interactive, almost too interactive. Because no tactical language exists for LP, it is impossible to add your own decision procedures or proof heuristics. A list of commands can be saved in a script file and executed again but this is no substitute for a tactical language. In LP it is impossible to express things like: 'Try different proofs till one succeeds' or to examine the structure of terms like 'If the current goal contains an if-then-else with a single variable as boolean, then resume by a case distinction on that variable'.

*Proof Management* The Proof Management system of LP is very simple. When LP generates subgoals in response to a case distinction or induction proof the order in which these subgoals must be proved is decided on by LP. The only way to escape from this rigid system is by adding a subgoal as an axiom to the system, continue with the next subgoal and leave the first one for another day. Then rerun the generated script up to the point where the "axiom" is added and then insert a prove for that subgoal. There is some danger in this method because LP lacks a special draft/proof mode switch. It is always allowed to add axioms, so it is up to the user to check that in the final version no "axioms" remain that need a proof.

It is also always allowed to cancel a proof. Obvious this is the quickest way to get at the qed, which technically means in LP that there are no conjectures left to prove. We claim that our proof does not contain unintended added axioms or cancels. We used `grep`[2] to search for these commands in our proof script files. Still we think that a special proof mode, that one can enter after loading the axioms, would give some extra confidence in LP proofs.

As mentioned before, the script file that contains our proof is about 150 Kb and takes 1:30 hours to run. Imagine that one makes a change at the end and wants to check if LP accepts the new script. This will take 1:30 hours for each revision. So to keep the proof manageable it is split in 60 lemmas and each lemma is proved in a separate file. So if we change the proof of one lemma we only have to check if LP accepts that one. All lemmas are listed in one file. By assigning a level to each lemma and requiring that only lower level lemmas are used we

[2]Unix command to search for a string in a set of files

ensure that there is no cycle in the proof. To make life a little easier a small `nawk`[3] program is used to generate the standard begin of the LP scripts. That is: a command to load the axioms, some settings and adding the lower level lemmas as axioms.

LSL comes with a library of traits. Most of these traits contain an `implies` clause that contains some important lemmas for that trait. But it comes without a proof, so the complete sceptic is not satisfied. The scriptfiles can be used to distribute proofs for these lemmas. Some care must be taken because LP proofs tend to be context dependent. Operationally LP proofs depend heavily on the normalization, and so on the set of facts and the direction in which the rules are oriented. Logically one can extend the set of facts without harming the proof, operationally a new rule can disturb a proof. This problem can be solved by making all rules inactive except the rules that are used. And then force the orientation, for example by giving a partial ordering on the function symbols. The generation scheme of fresh constants and variables is also a source of context dependency. For instance in a proof by induction or when assuming the left hand side of an implication, fresh variables and/or constants are generated. Variable names are b, b1, b2 ... for variables whose sort name begins with a B and the first free name is chosen. For constants a c is added so the names are: bc, bc1, bc2 .... Because these names are generated in a proof context the next proof does not know about these names. But the constants and variables that we declare at top level are visible everywhere, and so they can influence the generation of names. To prevent problems with unexpected names of fresh constants and variables it is advisable to use names that are not in the generation scheme of LP.

*Software Management* Besides that Larch is used to describe software, LP itself is a software product. As to be expected with an experimental tool as LP we encountered some bugs. A critical bug was an alpha conversion problem. When `n`,`m` and `k` are of type **Nat** and this is a rewrite rule: `n <= m -> \E k (n + k = m)`. The normalform of `n <= k` was according to LP (3.1 94/12/30): `\E n (n + n = k)`. In the current version of LP (3.1a 95/04/27) this bug is fixed and the normalform is: `\E n1 (n + n1 = k)`.

It was far more time consuming to cope with a memory problem. Even when memory on the machine and heap space in LP are ample available LP still runs out of memory. During our proof the system contains up to about four hundred rules and the proof is up to ten levels deeply nested. Steve Garland advised (via mail) to issue the command `forget` to delete a data structure used for completion of rewrite systems. This indeed frees memory, but not enough for our proof. Our "solution" was to delete rules that are not needed any more in the current proof context. Finding the "deletable" rules is a time consuming trial and error process. One deletes a lot of rules to find out later that a few of them are still needed, and then starts again this time without deleting those rules, this time LP runs out of memory, etc.

Just for the record: we eventually checked the entire proof with LP Release 3.1a (95/04/27).

Next we will give our wish list for LP and LSL. We think that the tactical language, for our purposes, is by far the most important wish.

[3]A C-like pattern matching language.

1. **Tactical language** Without a good tactical language it is impossible to extend LP with heuristics or decision procedures. So proofchecking with LP remains at the level of normalizing conjectures and proofs by induction, etc. While for really efficient use of proof-checkers it is necessary to have larger concepts. For instance a tactic like: try to proof this invariant by case distinction on all booleans and apply a decision procedure on the remaining expressions over the reals.

2. **Arithmetic decision procedures** With or without a tactical language, arithmetic decision procedures would be very useful. In [LSGL94] it is mentioned that a procedure for linear inequalities is implemented. Unfortunately most of the expressions in our proof are not linear.

3. **Larger proofs accepted without memory problems**

4. **Better proof management** It should be possible to use unproved lemmas and return to them later or skip cases of an induction proof etc. Furthermore it would be nice if a proof can be saved in such way that the proved theorem can be loaded directly, also when the current set of facts is extended compared to the set from which the theorem is proved.

5. **Explicit names in LSL specifications** Standard facts in LP have the name of the corresponding specification followed by a number, like in `Nat.3`. One can work around this because a conjecture can be named explicitly and a conjecture that is an axiom is easy to prove.

6. **More control over rewriting** Sometimes a term has different redexes, and it is useful to be able to select one by hand. For normalizing it would be nice if it is possible to influence the order in which the rewrite rules are applied.

### *7.3 Related Work*

The EEL protocol has received some attention from other sites. In [HWT95] Ho and Wong-Toi analysed the audio control protocol using the HyTech tool. HyTech is a symbolic model checker for linear hybrid systems. Larsen, Pettersson and Yi analysed the protocol with the UPPAAL tool [LPY95]. They used a formalization of the protocol based on the one developed by Ho and Wong-Toi. Daws and Yovine analysed the protocol using KRONOS in [DY95]. The formalization used in this paper is different from the one used by the two others. It is not completely clear what the formal relation is between the finite state description used by the model checkers and the version as presented in [BPV94]. It seems to be an interesting research problem how to integrate model-checking and proofchecking. In [MN95] Müller and Nipkow discuss this topic.

In [NS95] the I/O automata model is formalized in Isabelle/HOL. In that paper a much larger part of the I/O automata model is formalized, in contrast to the limited number of notions that are formalized in this paper.

Related to wish 4 is the work discussed in [Voi92]. In this paper a new proof environment for LP is proposed which makes it possible to walk through the proof tree as suggested in point 4.

## A. Example Proof

In this appendix a scriptfile of a LP proof is given. It is hard to read the proof because it depends on the set of facts, which changes with every proof command. So this proof is presented here to give some idea of what a scriptfile looks like more than to show the actual proof.

```
set name p
prove :timetwLT: y <= 8 /\ x # y => x < (y + 1)
  res by =>
    <> => subgoal
    prove :step9: (2*yc) <= 16
      ins x by yc, y by 8, z by 2 in timeF2
      [] conjecture
    prove :step8: ((2 * yc) + 1) <= 17
      ins x by (2 * yc), y by 16, z by 1 in timeF1
      [] conjecture
    prove :step7: ((2 * yc) + 1) < d(T)
      prove :hulpje: 17 < d(T)
        ins x by T, y by d(17), z by d(T) in TimeLTF2
        ins x by T in time5
        ins x by 1, y by (d(17) * d(T)), z by 17 in timeLTF2
        [] conjecture
      ins x by ((2 * yc) + 1), y by  17, z by d(T) in timeLTran1
      [] conjecture
    prove :step6: ((2 * yc * T) + T) < 1
      ins x by ((2 * yc) + 1), y by d(T), z by T in timeLTF2
      make ina timeDis
      rew timeLTF2 with rev timeDis
      [] conjecture
    prove :step5: (2 * yc * T) < (1 - T)
      ins x by ((2 * T * yc) + T), y by 1, z by m(T) in timeLTF1
      [] conjecture
    prove :step4: (yc * T) < (1 + m(yc * T) + m(T))
      ins x by (2 * T * yc), y by (1 + m(T)), z by m(T * yc) in timeLTF1
      prove (2 * T * yc) + m(T * yc) = (T * yc)
        set im on
        prove :hulpje: 2 = 1 + 1
          [] conjecture
        make ina hulpje
        rew con with rev hulpje
        make ina timeDis
        rew con with rev timeDis
        rew con with rev timeDis
        [] conjecture
      [] conjecture
    prove :step3: (yc + (yc * T)) < (yc + (1 + m(T + (yc * T))))
      ins x by (yc * T), y by (1 + m(T + (yc * T))), z by yc in timeLTF1
      [] conjecture
    prove :step2: ((1+T)*yc) < ((yc +1)* (1-T))
      make ina timeDis
      rew con with rev timeDis
      rew con with rev timeDis
      rew con with rev timeDis
      [] conjecture
    prove :step1: ((1+T) * d(1-T) * yc) < (yc + 1)
      ins x by ((1 + T) * yc), y by ((yc + 1) * (1 - T)), z by d(1-T) in timeLTF2
      ins x by (1- T) in time5
      [] conjecture
    prove :step0: xc < (yc + 1)
      set im anc
      ins x by yc, y by xc in TimeDefTwiddle
      ins x by xc, y by yc in timecom
      ins x by xc, y by ((1 + T) * d(1 + m(T)) * yc), z by (yc + 1) in timeLTran1
      [] conjecture
    [] => subgoal
  [] conjecture
%% quit
```

## B. The traits

### *B.1 S*

In this section we will present the trait that defines System S piece by piece.

```
                                                                              1
S : trait
  includes System(S), List, CommonActions(S)
    States[S] tuple of S: Send, R : Rec, error : Bool
    Send tuple of transmitting: Bool,
                  wire_high   : Bool,
                  list        : List,
                  x           : Time
    Rec tuple of  list        : List,
                  x           : Time
```

Above the first lines of the trait are given. The name of the trait is S, given on the first line of the trait. Then the trait System(S) (see Section 4.2) and the traits List and Time are included, see Section 4.1. Next the sort `States[S]` is defined, its domain consists of triples of (1) the state variables of the sender, (2) the state variables of the receiver and (3) a history variable `error`. A history variable does not influence the behaviour of the system. The extra information it provides is only used in the proof. The sender has four state variables: `transmitting` is true when the system is transmitting, that is from the first *UP* action till the last *DOWN* action. The variable `wire_high` denotes the level of the voltage on the bus. The variable `list` contains the bits of the message that still must be transmitted. The clock variable `x` is used to specify the distance between the UP and DOWN actions of the sender. The receiver has two state variables: `list` denotes the bits of the current message that are already received and the clock variable `x` denotes the time elapsed since the last UP action.

```
                                                                              2
  introduces
    UP   :                              -> Actions[S]
    DOWN :                              -> Actions[S]
    aux  : States[S], Time, Actions[S] -> Bool           %% auxiliary-function.
```

The UP and DOWN actions are constants of type `Actions[S]`, the other actions IN, OUT and TIME are declared in `CommonActions` trait. Furthermore the auxiliary function `aux` is declared, which is used in the action-predicate of the TIME action.

3
```
  asserts
    Actions[S] generated by IN, UP, DOWN, OUT, TIME
    \forall s,s': States[S], m : List, t,x,y,z: Time
```

After the `asserts` key-word the properties of the functions (the axioms) are given. The `generated by` clause expresses that every action is an IN, UP, DOWN, OUT or TIME action.

4
```
  ~isVisible(UP);
  ~isVisible(DOWN);
```

The actions UP and DOWN are declared invisible, the actions IN, OUT and TIME are declared visible in the `CommonActions` trait.

5
```
%% *** START STATES ***
start(s) = (   ~s.S.transmitting
            /\ ~s.S.wire_high
            /\ s.S.list = empty
            /\ ~s.error
            /\ s.R.list = empty);
```

Initial the system is not transmitting, the wire is low, there is no message in transit (the lists are empty) and no error has occurred yet. Note that the values of the clocks (S.x and R.x) are undefined and so we have an infinite number of start states.

6
```
%% *** IN(m) ***
pre(s, IN(m)) =
     (head(m) = 1 /\ (odd(length(m)) \/ last_two(m) = [0] ^ [0]));
eff(s, IN(m), s') = (
     (if ~s.S.transmitting /\ ~s.S.wire_high /\ s.S.list = empty
         then s'.S.list = m /\ s'.S.x = 0
         else s'.S.list = s.S.list /\ s'.S.x = s.S.x)
  /\ s'.S.transmitting = s.S.transmitting
  /\ s'.S.wire_high = s.S.wire_high
  /\ (if s.R.list ~= empty then s'.error else s'.error = s.error)
  /\ s'.R = s.R) ;
```

The IN action denotes the reception of a new message to be transmitted. The precondition expresses that each message must start with a bit 1 and that a message must be of odd length or end in $< 00 >$ (see Section 2). The `error` variable becomes true when an IN action occurs too early, that is, when the receiver is not yet ready to receive a new message.

7

```
%% *** UP ***
pre(s,UP) = aux(s,s.S.x,UP);
aux(s,x,UP) = (
     ~s.S.wire_high
  /\ s.S.list ~= empty
  /\ (if s.S.transmitting
         then (if head(s.S.list)=1 then x = 4 else x = 2)
         else x = 0));
eff(s,UP,s') = (
     s'.S.transmitting
  /\ s'.S.wire_high
  /\ (if head(s.S.list) = 1
         then s'.S.list = tail(s.S.list) /\ s'.S.x = 0
         else s'.S.list = s.S.list /\ s'.S.x = s.S.x)
  /\ s'.error = s.error
  /\ (s.R.list = empty  => s'.R.list = [1])
  /\ (last(s.R.list) = 0 /\ s.R.list ~= empty =>
             (s.R.x < 3               => s'.R.list = empty)
          /\ (3 <= s.R.x /\ s.R.x < 5 => s'.R.list = s.R.list ^ [0])
          /\ (5 <= s.R.x              => s'.R.list = s.R.list ^ [0] ^ [1]))
  /\ (last(s.R.list) = 1 =>
             (s.R.x < 3               => s'.R.list = empty)
          /\ (3 <= s.R.x /\ s.R.x < 5 => s'.R.list = s.R.list ^ [1])
          /\ (5 <= s.R.x /\ s.R.x < 7 => s'.R.list = s.R.list ^ [0])
          /\ (7 <= s.R.x              => s'.R.list = s.R.list ^ [0] ^ [1]))
  /\ s'.R.x = 0);
```

The UP action corresponds to an up-going edge on the wire. The sender generates the UP's and DOWN's as required by the Manchester encoding. The receiver's algorithm to decode the message via the UP actions is a direct formalization of the algorithm in Philips' documentation.

8

```
%% *** DOWN ***
pre(s,DOWN) = aux(s,s.S.x,DOWN);
aux(s,x,DOWN) = (
    s.S.wire_high
 /\ (if s.S.list ~= empty /\ head(s.S.list) = 0 then x = 4 else x = 2));
eff(s,DOWN,s') = (
     (if s.S.list = empty \/ s.S.list = [0]
         then ~s'.S.transmitting
         else s'.S.transmitting = s.S.transmitting)
  /\ ~s'.S.wire_high
  /\ (if s.S.list ~= empty /\ head(s.S.list) = 0
         then s'.S.list = tail(s.S.list) /\ s'.S.x = 0
         else s'.S.list = s.S.list /\ s'.S.x = s.S.x)
  /\ s'.error = s.error
  /\ s'.R = s.R);
```

The DOWN action of course corresponds to a down-going edge on the bus. The receiver does not observe this action, and indeed the values of the state variables of the receiver do not change (`s'.R = s.R`).

```
%% *** OUT ***                                                              9
pre(s,OUT(m)) =
      (m = finalize(s.R.list) /\ aux(s,s.R.x,OUT(m)));
aux(s,x,OUT(m)) = (
      s.R.list ~= empty
  /\ (if last(s.R.list) = 0 then x = 7 else x = 9));
eff(s,OUT(m),s') = (
     s'.R.list = empty
  /\ s'.R.x = s.R.x
  /\ s'.S = s.S
  /\ s'.error = s.error);
```

When the receiver received the complete message the OUT action happens. Hereafter the receiver is ready to receive a new message.

```
%% *** TIME ***                                                             10
isStep(s,TIME(t), s') = (
     t > 0
  /\ s'.S.transmitting = s.S.transmitting
  /\ s'.S.wire_high = s.S.wire_high
  /\ s'.S.list = s.S.list
  /\ s'.error = s.error
  /\ (1-T) <= ((s'.S.x - s.S.x) / t) /\ ((s'.S.x - s.S.x) / t) <= (1+T)
  /\ ((\E y (y >= s.S.x /\ aux(s,y,UP))) =>
                   (\E z (z >= s'.S.x /\ aux(s',z,UP))))
  /\ ((\E y (y >= s.S.x /\ aux(s,y,DOWN))) =>
                   (\E z (z >= s'.S.x /\ aux(s',z,DOWN))))
  /\ s'.R.list = s.R.list
  /\ (1-T) <= ((s'.R.x - s.R.x) / t) /\ ((s'.R.x - s.R.x) / t) <= (1+T)
  /\ ((\E y (y >= s.R.x /\ aux(s,y,OUT(m)))) =>
                   (\E z( z >= s'.R.x /\ aux(s',z,OUT(m))))))
```

In the TIME action the discrete variables (**`transmitting`**, **`wire_high`**, **`list`**, **`error`**) are not allowed to change. Only the clocks are advanced by "about" t, formally: for both clocks $(1-T) \leq \frac{x'-x}{t} \leq (1+T)$ must hold.

## *B.2 P*

In this appendix the correctness criterion is given. It is a LSL-version of system P of [BPV94].

```
P : trait
  includes System(P), Time, List, CommonActions(P)
    States[P] tuple of list  : List,
                       chaos : Bool,
                       x     : Time
  introduces
    aux  : States[P], Time, Actions[P] -> Bool           %% auxiliary-function

  asserts
```

```
    Actions[P] generated by IN, OUT
    \forall s,s' : States[P], t : Time, m : List, x,y,z: Time

start(s) = (s.list = empty /\ not(s.chaos));

pre(s,IN(m)) = (
     head(m) = 1
  /\ (odd(length(m)) \/ last_two(m) = [0] ^ [0]));
eff(s,IN(m),s') =
     (if s.list = empty
         then (s'.list = m /\ s'.x = 0 /\ s'.chaos = s.chaos)
         else (s'.chaos = true /\ s'.list = s.list /\ s'.x = s.x));

pre(s,OUT(m)) = aux(s,s.x,OUT(m));
aux(s,x,OUT(m)) = (
    (   s.list = m
     /\ s.list ~= empty
     /\ ((1-T)*x) <= ((4 * t(length(s.list))) + 5))
 \/ s.chaos);
eff(s,OUT(m),s') =
     (s'.list = empty /\ s'.chaos = s.chaos /\ s'.x = s.x);

isStep(s,TIME(t),s') = (
    t > 0
 /\ s'.list = s.list
 /\ s'.chaos = s.chaos
 /\ s'.x - s.x = t
 /\ \A m ((\E y (y >= s.x /\ aux(s,y,OUT(m)))) =>
                    (\E z( z >= s'.x /\ aux(s',z,OUT(m))))))
```

*B.3 CommonActions*

The trait given in this appendix is needed for technical reasons. In LSL sorts are disjunct and here a `CommonActions` sort is introduced to make it possible to compare actions of different systems.

```
CommonActions(A) : trait
  assumes System(A)
  introduces
    IN         : List -> Actions[A]
    OUT        : List -> Actions[A]
    TIME       : Time -> Actions[A]
    IN         : List -> CommonActions
    OUT        : List -> CommonActions
    TIME       : Time -> CommonActions
  asserts \forall m: List, t : Time
    common(IN(m))   = IN(m);
    common(OUT(m))  = OUT(m);
    common(TIME(t)) = TIME(t);
    isVisible(IN(m));
    isVisible(OUT(m));
    isVisible(TIME(t))
```

### *B.4 Bit*

In this appendix a little trait for the sort Bit is given. Bits are the elements of the messages transmitted by the protocol.

```
Bit : trait
introduces
  0,1 : -> Bit
asserts \forall bit: Bit
  (bit = 1) \/ (bit = 0);
  1 ~= 0
```

### *B.5 Nat*

The sort Nat is introduced because it is the result sort of the `length` function on lists.

```
Nat : trait
  introduces
    0 : -> Nat
    s : Nat -> Nat
    __ + __ : Nat, Nat -> Nat
    odd: Nat -> Bool
  asserts
  Nat generated by 0,s
   \forall n,n1: Nat
    0 ~= s(n);
    s(n) = s(n1) <=> n = n1;
    n + 0 = n;
    n + s(n1) = s(n + n1);
    ~odd(0);
    odd(s(n)) = ~odd(n)
```

### *B.6 Time*

Here the `Time` trait is given. It is discussed in section 4.1.

```
Time : trait
  includes Bit, AC(+,Time), AC(*,Time), Nat
  introduces
    __ <= __, __ >= __ : Time, Time -> Bool
    __ < __ , __ > __  : Time, Time -> Bool
    __ + __, __ - __ : Time, Time -> Time
    __ * __, __ / __ : Time, Time -> Time
    m : Time -> Time
    d : Time -> Time
    min : Time , Time -> Time
    t : Bit -> Time
    t : Nat -> Time
    0,1,2,3,4,5,6,7,8,9 : -> Time
    10,11,12,13,14,15,16,17,18,19,20 : -> Time
    __ # __ : Time,Time -> Bool
    T,pTmT,mTpT : -> Time
  asserts \forall x,y,z,t: Time, n : Nat
%% Studies in Logic and the Foundations of Mathematics. (Chang, C. C. and Keisler, H. J.)
%% Volume 73 Model Theory (Page 37)
%% LINEAR ORDER Axioms.
    x <= y /\ y <= z => x <= z;    %% TimeL1
    x <= y /\ y <= x => (x = y);   %% TimeL2
    x <= x;                        %% TimeL3
    x <= y \/ y <= x;              %% TimeL4
%% ABELIAN GROUPS.
    %%  x + (y + z) = (x + y) + z; %%         (associativity of +)
    x + 0 = x;                     %% TimeId  (identity)
    %%  \E y (x + y = 0 /\ y + x = 0);%%
    x + m(x) = 0;                  %% TimeMx  (existence of inverse)
    %%  x + y = y + x;             %%         (commutativity of +)
%% FIELDS = These 6 axiomas + Abelian groups.
    1 * x = x;                     %% TimeUn  (1 is unit)
    %%  x * (y * z) = (x * y) * z; %%         (associativity of *)
    %%  x * y = y * x;             %%         (commutativity of *)
    (x * y) + (x * z) =  x * (y + z);%% TimeDis (distributivity of * over +)
    0 ~= 1:Time;                   %% Time10
    %% x ~= 0 => \E y (y * x = 1); %%         (existence of multiplicative inverse.)
    x ~= 0 => (x * d(x) = 1);      %% TimeDx
%% ORDERD FIELDS = These 2 + Field + Linear Order.
    x <= y => (x + z) <= (y + z);  %% TimeF1
    x <= y /\ 0 <= z => (x * z) <= (y * z); %% TimeF2

    t(0:Bit) = 0;
    t(1:Bit) = 1;
    t(0:Nat) = 0;
    t(s(n)) = t(n) + 1;

%% Notational convenience.
    x >= y             = y <= x ;
    (x <= y /\ x ~= y) = x < y;     %% TimeLT
    x > y              = y < x ;
    x/y                = x * d(y);
    x - y              = x + m(y);
    min(x,y)           = (if x <= y then x else y);%% TimeMin

     1 + 1 =  2;  2 + 1 =  3;  3 + 1 =  4;  4 + 1 =  5;  5 + 1 =  6;
     6 + 1 =  7;  7 + 1 =  8;  8 + 1 =  9;  9 + 1 = 10; 10 + 1 = 11;
    11 + 1 = 12; 12 + 1 = 13; 13 + 1 = 14; 14 + 1 = 15; 15 + 1 = 16;
    16 + 1 = 17; 17 + 1 = 18; 18 + 1 = 19; 19 + 1 = 20;


%% Specific for this proof:
0 < T /\ T < (1/17) ;
pTmT = ((1+T)/(1-T)); mTpT = ((1-T)/(1+T));
((((1-T) / (1+T)) * x) <= y /\ y <= (((1+T) / (1-T)) * x)) = x # y
```

## References

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 1992.

[BPV94] D.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an audio control protocol. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the Third International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems,* Lübeck, Germany, September 1994, volume 863 of *Lecture Notes in Computer Science*, pages 170–192. Springer-Verlag, 1994. Full version available as Report CS-R9445, CWI, Amsterdam, July 1994.

[CL92] B. Chetali and P. Lescanne. An exercise in LP: The proof of the non restoring division circuit. In *First International Workshop on Larch, Dedham*, pages 55–68. Workshops in Computing, Springer-Verlag, July 1992.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. Technical Report Spectre-95-06, VERIMAG, Grenoble, France, April 1995. Also appeared in *Proceedings 2nd European Workshop on Real-Time and Hybrid Systems*, June 1995, Grenoble, France.

[GG] S.J. Garland and J.V. Guttag. LP: Introduction. `http://larch-www.lcs.mit.edu:8001/larch/LP/overview.html`.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Report 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Gri94] W.O.D. Griffioen. *Analysis of an Audio Control Protocol with Bus Collision.* Master thesis, Programming Research Group, University of Amsterdam, 1994.

[GSSL93] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA, December 1993.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *Proceedings of the 7th International Conference on Computer Aided Verification,* Liège, Belgium, Lecture Notes in Computer Science. Springer-Verlag, July 1995.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic model-checking for real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, oct 1995. Springer Verlag, Lecture Notes in Computer Science.

[LSGL94] V. Luchangco, E. Söylemez, S. Garland, and N.A. Lynch. Verifying timing properties of concurrent algorithms. In *Proceedings of the Seventh International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 239–259, Berne, Switzerland, October 1994. IFIP WG6.1, Elsevier Science Publishers B. V. (North Holland). Preliminary version. Final version to be published by Chapman and Hall.

[LV92] N.A. Lynch and F.W. Vaandrager. Action transducers and timed automata. In W.R. Cleaveland, editor, *Proceedings CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*, pages 436–455. Springer-Verlag, 1992. Full version available as CWI Report CS-R9460, Amsterdam, November 1994, and as Technical Memo MIT/LCS/TM-480.b, MIT LCS, Cambridge, MA, October 1994. To appear in *Formal Aspects of Computing.*

[LV93] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part II: Timing-based systems. Report CS-R9314, CWI, Amsterdam, March 1993. Also, MIT/LCS/TM-487.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. To appear in *Information and Computation.*

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations. part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Aarhus, Denmark, volume NS-95-2 of *BRICS Notes Series*, pages 1–12. Department of Computer Science, University of Aarhus, May 1995.

[NS95] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proceedings International Workshop TYPES'94*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[Voi92] Frédéric Voisin. A new front-end for the larch prover. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch, Dedham 1992*, pages 282–296. Workshops in Computing, Springer-Verlag, 1992.