Formal Verification of a Protocol for Communications over faulty channels

Boutheina Chetali & Pierre Lescanne
CRIN-CNRS and INRIA-lorraine
University of Henri Poincaré,Nancy I, Campus Scientifique —B.P. 101, 54602 Villers-les-Nancy — FRANCE
email: `chetali@loria.fr, lescanne@loria.fr`

Abstract
This paper describes the formalization and the verification of a communication protocol over faulty channels, using the Larch prover LP. The chosen specification environment is UNITY, since the proposed model can be fruitfully applied to a wide variety of problems and modified or extended for special purposes. It provides a higher level of abstraction to express solutions to parallel programming problems. We show how the UNITY specifications can be mechanized in LP, how we can use the theorem proving methodology to prove safety and liveness of a communication protocol, and that a theorem prover can be actually used to detect flaws in a system specification.

Keywords
Formal Verification and Protocol Verification, Communication Protocol, theorem prover methodology, LARCH PROVER, UNITY logic and specification, computer checked proof.

1 INTRODUCTION

Several approaches to program verification have been proposed and used in the literature. Among them, there are two principal views, one operational and the other syntactic. The first one consists in analyzing the program in terms of its execution sequences, analysis which is often successful in the case of sequential programs but much less so in the case of concurrent programs. The syntactic approach is based on an axiomatic reasoning, where one needs a formalism to express the relevant properties of a program, an appropriate language to construct well-founded formulas and a proof system to construct proofs. The UNITY environment [Chandy and Misra, 1988] provides those tools to design and prove concurrent programs.

Despite the relative simplicity of the proposed model, the formal verification of a program specified in UNITY requires a careful analysis and involves an enormous amount of clerical detail. Therefore, a theorem prover is required to provide a very high degree of confidence in the correctness of the verification.

UNITY offers a logic and a notation in which abstract specification of the computation and its successive refinements are expressed, without any mention to execution sequences. This *static* view of the program recall the LARCH style of specification [Guttag *et al.*, 1993], which emphasizes brevity and clarity rather than executability.

In this paper, we describe a formal proof of a concurrent program specified in UNITY. We show how the UNITY specifications can be mechanized in LP, and how we can use the theorem proving methodology to prove safety and liveness of a communication protocol. For that, we present the fully computer checked proof of a program describing a protocol for communications over faulty channels. Our aim is to show that we can take advantage of both the power and the simplicity of UNITY and LP, and that a theorem prover can be actually used to detect flaws in a system specification, even simple one presented in textbooks.

The protocol studied in this paper is taken from Chapter 17 of the textbook of Chandy and Misra [Chandy and Misra, 1988]. The given protocol is a generalization of the Alternating Bit Protocol. Despite its simplicity, it contains all the difficulties and problems of a real protocol and allows us to easily illustrate the solutions we propose. Even in such a basic protocol, we found bugs and flaws, essentially two in our experiments.

First for the safety properties, the full mechanization of the proof lead us, as often in such an experiment, to enter low-level detail of the specification. This necessary refinement forces us to address assertions whose proof is either understood or left to the reader, and as expected we found there underestimation of the actual difficulty of the proof. Precisely Chandy and Misra's proof is based on the invariance of three predicates I_1, I_2 and I_3 and the authors understood that the invariance of each predicate can be proved independently. The experiment with LP has shown that the invariance of I_1 requires this of I_3 and this of an additional predicate.

Second for the liveness properties, we highlighted the fact that the low-level steps *left to the reader* carry the deeper aspect of the protocol, e.g. the data communication over the faulty channels. Moreover the experiment has taught us that the strategy aspect in developing a proof of liveness cannot at all be neglected.

The paper is organized as follows. We start by a brief description of the fundamental concepts of UNITY. Then we describe the communication protocol provided by Chandy and Misra [Chandy and Misra, 1988], and its formalization in LP. Next we show the main steps of the mechanization of the correctness proof. In particular, we explain the flaws detected in the safety proofs, and we describe the strategy used to prove progress properties. Finally, we conclude with a discussion about this study and future work.

We suppose a little familiarity with the LP system (as few as possible) and we recommand [Garland and Guttag, 1991] as an introduction. This experiment has been conduct with the release 3.1 of LP.

2 ABOUT UNITY

For our purpose, a UNITY program consists of three sections: a **declare** section that declares the variables used in the program, an **initially** section that describes the initial values of the variables, and an **assign** section that consists of a non-empty set of assignment statements. Execution of every UNITY statement (assignment) terminates in every program state and a program execution consists of an infinite number of steps in which each statement is executed infinitely often. A UNITY program terminates by reaching a *fixed point*, which is equivalent to termination in standard sequential-programming terminology. The model proposed by UNITY is traditional in the sense that it is a state-transition system with named variables to express the state and conditional multiple assignment to express state transitions. The simplicity of the model comes from the absence of communication construct and flow control.

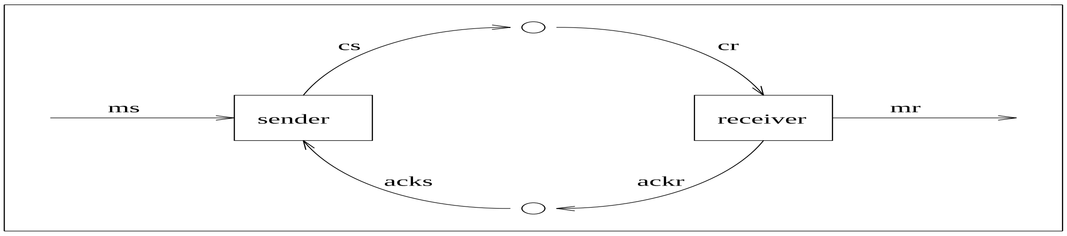

Figure 1 The protocol system components.

Correctness of concurrent programs is defined in terms of properties of execution sequences. There are two basic kinds of properties of concurrent programs: *Safety*, the property must always be true, and *Liveness*, the property must eventually be true [Lamport, 1977]. There are five relations on predicates in UNITY theory: *Unless*, *Stable*, *invariant*, *Ensures* and *leads_to*. The first three are used for stating safety properties whereas the last two are used to express *progress* properties, a subset of liveness properties.

The reader can find the specifications of the UNITY logic and methodology in LP in [Chetali, 1995a], with the encoding of the syntax of UNITY, the wp-calculus and the syntax of first order predicates. We give in the appendix only the definitions needed in this paper together with some hints about LP.

3 A PROTOCOL FOR COMMUNICATION OVER FAULTY CHANNELS

In this section, we describe the fully computer checked proof of the protocol for communications over faulty channels proposed as a case study by Chandy and Misra in [Chandy and Misra, 1988]. The faulty channel and the protocol are specified as program specifications. Thus the task of designing a communication protocol over faulty channels is viewed as a program composition: the union of the protocol program and the faulty-channel program is a program which guarantees fault-free communication. In order to keep the length of the example within a reasonable size, only the basic ideas are sketched here.

A process called the *sender* has access to an infinite sequence of data, ms. Another process, the *receiver*, is required to output a sequence mr satisfying the following specification, where $|\, mr \,|$ is the length of mr:

invariant $mr\ is\ a\ prefix\ of\ ms$
$$|mr| = n \quad leads\_to \quad |mr| = n + 1$$

The sender and the receiver communicate through a faulty channel modeled by two pairs of nonfaulty channels (cs, cr) and $(ackr, acks)$, and a program that manipulates the contents of cs and cr (Fig 1). The sender adds messages to cs and receives acknowledgments from $acks$, the receiver receives message from cr and adds acknowledgments to $ackr$. The sender *writes* a data item and its index as a pair $(ks, ms[ks])$ in cs. So at any moment, ks represents the number of the last message sent by the sender. In the same way, the receiver receives from cr a pair (n, d). So it adds the data item d to mr and sends n as an acknowledgment to the sender (using the channel $ackr$). At each time, kr is the number of the last message received by the receiver. Thus the sender sends a new message only upon receiving an acknowledgment for the previous one (see *Invariant* I_1 given below).

Specification of the faulty channel:

It is important to specify a faulty channel such that lost or duplication of messages are not under the control of the sender and the receiver, but may be done autonomously. **Safety** properties state that a faulty channel can only remove the head of cs or append to the rear of cr; furthermore, a faulty channel may loose messages from cs and/or duplicate messages in cr. A **progress** specification of a faulty channel is that, if a message m is appended infinitely often to cs and no other message is appended to cs, then m appears in cr eventually. The corresponding specification of the faulty channel could be found in [Chandy and Misra, 1988].

Chandy and Misra give the UNITY program, *FC*, which represents a faulty channel and the proof of its correctness. This demonstration is not necessary for the study of communication protocols; only the specification of a faulty channel is required to develop and verify protocols, thus we assume the correctness of the proof of the faulty channel and we take its specification as axioms in our proof. It is important to note that this work is not intented to be a proof of the adequacy of the formalization (modelization) of the communication protocol but the proof of the correctness of the protocol in the framework of UNITY. Readers who want to learn more about the formalization are referred to [Chandy and Misra, 1988]. Indeed, we focus our study on the correctness proof of the program which specify the protocol. We show, using LP, that the proposed proof is incomplete.

The protocol
A communication protocol is a program which fulfills the requirements of a correct communication when it is composed with the program (*FC*) which meets the requirements of a faulty channel.

Program {Protocol}
Declare $ks, kr : integer$
initially
$ks, kr = 1, 0$ [] $cs, cr, acks, ackr = null, null, null, null$ [] $mr = null$
assign

$\{transmit\}$	cs	$:= cs; (ks, ms[ks])$	
$[]\{rcv\_ack\}$	$ks, acks$	$:= ks + 1, tail(acks)$	$if\ acks \neq null \wedge ks = head(acks)$
$[]\{rmv\_old\_ack\}$	$acks$	$:= tail(acks)$	$if\ acks \neq null \wedge ks \neq head(acks)$
$[]\{receive\}$	kr, mr, cr	$:= kr + 1, mr; head(cr).val, tail(cr)$	$if\ cr \neq null \wedge kr \neq head(cr).dex$
$[]\{send\_ack\}$	$ackr$	$:= ackr; kr$	
$[]\{rmv\_old\_msg\}$	cr	$:= tail(cr)$	$if\ cr \neq null \wedge kr = head(cr).dex$

end{Protocol}

This program is slightly different from the one in [Chandy and Misra, 1988], due to the not encoding of the parallel assignment ||. The two programs are equivalent under this modification. The first three assignments statements constitute the sender and the last three the receiver.

The first action, *transmit*, in the sender sends a new message by appending the pair $(ks, ms[ks])$ to the transmission channel cs. The second action, *rcv_ack*, models the receipt of acknowledgment, e.g the value in the head of the acknowledgment channel is equal to the number of the last message sent

$(ks = head(acks))$. The action *rmv_old_ack* takes care of re-transmission. As $(ks \neq head(acks))$, the value of ks is not changed and the head of the acknowledgment channel is removed (it is an old acknowledgment).

The first action, *receive* in the receiver corresponds to a receipt of a new message, e.g the value in the head of the transmission channel differs from the number of the last message received $(kr \neq head(cr))$. The delivery of the message is modeled by the removal of the head of the transmission channel and the appending of the data part of the message to the sequence of data ms. The action *send_ack* models the sending of the acknowledgment, and the last one, *rmv_old_msg*, models the removal of the head of the transmission channel if it contains an old message $(kr = head(cr))$.

Note: The Alternating Bit protocol is an optimization of this protocol, e.g when the index parts of the messages are computed modulo 2.

Correctness of the protocol: To prove the correctness of the protocol, the following invariants and the progress property have to be proved.

Invariant
$$\begin{array}{ll} \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\ \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle & \wedge \\ kr \leq ks \leq kr+1 & \end{array} \qquad (I_1)$$

Invariant
$$\begin{array}{ll} \langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle & \wedge \\ mr\ is\ a\ prefix\ of\ ms & \wedge \\ \mid mr \mid = kr & \end{array} \qquad (I_2)$$

Invariant $cs.dex, cr.dex, acks, ackr\ are$ sequences of nondecreasing integers in Protocol. (I_3)

$$|mr| = n \quad \textbf{leads\_to} \quad |mr| = n+1 \qquad (P)$$

The first invariant expresses requirements on the integer part of the messages in the transmission channel and in the acknowledgment channels. The second one says that all message which are delivered by the receiver must have been sent by the sender and the third one is a property on the acknowledgments. The progress property states that all messages which the sender sends eventually are received by the receiver.

4 REPRESENTATION IN LP

Much of the power of the deductive system of LP relies upon *normalization*, e.g. computation of normal forms by rewriting. The basis for proofs is a logical system, which consists of signatures, equations, rewrites rules, operators theories etc. In the following, we explain the main steps in the translation of the UNITY program *protocol* in LP. We will present the notation exactly as it appears in LP since it might be interesting for the reader to see the formulae as they are seen at the screen when working with LP.

UNITY variables are of two kinds. The *identifiers* appear in the program text, they are *state* or *dynamic* variables (so called because they represent quantities that can vary with time). These variables are encoded as constant in LP of sort `Id`. The *proof variables* appear in the UNITY proof and are encoded as LP variables of sort `id_of_var`*.

A UNITY program is defined as a list of actions and encoded in LP as a constant of sort `Actlist`. The program *protocol* is then formalized by a constant `proto: -> Actlist`. A boolean expression `init` (a constant of sort `Bexp`) represents the **initially** section of the program *protocol*:

```
assert
init = (id_to_exp(kr) \= nat_to_exp(0))
       ^ (id_to_exp(ks) \=  nat_to_exp(succ(0)))
       ^ (seq_to_exp(id_to_seq(ackr)) \= seq_to_exp(emptyseq))
       :
```

The term `id_to_exp(kr) \= nat_to_exp(0)`represents the initial condition $kr = 0$, the function `id_to_exp` embeds the sort `Id` in the sort `exp` and `seq_to_exp` transforms a `sequence` into a `expression`. The term `(seq_to_exp(id_to_seq(ackr)) \= seq_to_exp(emptyseq))` formalizes the initial condition $ackr = null$. Identifiers and naturals are themselves expressions. As the logic of LP does not have subsorts, we explicitly embedded `Id`, `Id_of_var` and `nat` into `exp` using functions `id_to_exp` and `nat_to_exp`.

The **assign** section is encoded as a list of actions: proto = cons(*transmit*,...,cons(*rmv_old_msg*, nil)). For example, the actions *rcv_ack* is translated in LP using the function `cond_mult_assg(l,bexp)` which encodes multiple conditional assignments:

```
cond_mult_assg({ks.(id_to_exp(ks) + nat_to_exp(1))}
                 \u ({acks.seq_to_exp(tail(id_to_seq(acks)))}\u {}),
             nnot(id_to_exp(acks)\=seq_to_exp(emptyseq))
             ^(id_to_exp(ks) \= head(id_to_seq(acks))))
```

In the following, we do not write the embedding functions. For example, an expression of the form `id_to_exp(ks)\=head(id_to_seq(acks))` will be written `ks\=head(acks)`.

5 PROOF OBLIGATIONS

5.1 Safety Properties

The proof obligations describe properties that the protocol must satisfy. They are the equational axiomatization of the three invariants I_1, I_2, I_3. We expect that the *protocol* program satisfies the following properties:

- The transmit channels must not contain a message which has not been sent:
  $(x \in cr \vee x \in cs) \Rightarrow x.dex \leq ks$
- The receiver must only receive transmitted messages:
  $(x \in cr \vee x \in cs) \Rightarrow kr \leq x.dex$

*The difference in the encoding of state variables and proof variables comes from the fact that the former are involved in substitution. Substitutions are needed to encode the wp-calculus [Chetali, 1995a]

$\Rightarrow$ The receiver receives only what has been sent: $(kr \leq ks)$

- The acknowledgment channels must not contain an acknowledgment of a message which has not been received: $(y \in acks \vee y \in ackr) \Rightarrow y \leq kr$
- The receiver receives only what has been sent: $(kr \leq ks)$

$\Rightarrow$ The acknowledgment channels must not contain an acknowledgment of a message which has not been sent: $(y \in acks \vee y \in ackr) \Rightarrow y \leq ks$

Consider now the situation where a acknowledgment y is present in the acknowledgment channels, what could be the value of the last message sent (the value of ks)? It depends on the "kind" of the acknowledgment y: If it is an old acknowledgment, the value of ks is $y+1$ and y if not. As the sender sends a new message only upon receiving an acknowledgment for ks, y equals $ks-1$ or ks:
$(y \in acks \vee y \in ackr) \Rightarrow ks \leq y+1$
Finally, at any moment, kr, the number of the last message received is equal to the number of the last message sent ks or to the number of the previous one: $kr \leq ks+1$.

These requirements are formalized by the property I_1:

$$\textbf{Invariant} \quad \begin{array}{ll} \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\ \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle & \wedge \\ kr \leq ks \leq kr+1 & \end{array}$$

To prove this invariant in our system, we must prove the following theorem in LP:

```
Inv(Ib, proto, init)  = true
```

To express the predicate `Ib`, we define the predicate `inseq(e,seq)` which checks if the element `e` belongs to the sequence `seq`. The universal quantification on y and x is implicitly asserted using variables `yvar` and `xrec` of sort `id_of_var`. For example, the first conjunct of the boolean expression `Ib`
$\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle$
is expressed in LP as follow:
`[inseq(yvar,ackr) \or inseq(yvar,acks)] |=> [(yvar <= kr)^(ks <= (yvar+1))]`
An invariant property *invariant* P is proved to be satisfied by a program by proving that the initial condition establish P and that all the actions in the program maintain it, e.g. that the assertion $\{P\}s\{P\}$ holds for all assignments of the program. Let us describe the LP proof of I_1 using the *rcv_ack* action.

Proof of I_1:

(1): `inv(Ib, cons(rcv_ack,nil),init) = true`

LP expands this equation using the definitions of `invariant` and `unless`, introducing `wp` (*See the appendix*) :

(1): `((init \=> Ib) = T) /\ ((Ib |=> wp(rcv_ack,Ib)) = T)`

The command `res by /\-m` directs LP to use the conjunction method. The goal is split in two subgoals:

```
(1.1): init \=> (A ^ B ^ C) = T
(1.2): (A ^ B ^ C) \=> wp(cond_mult_assg({(ks,ks+1),(acks,tail(acks))},
                                  (ks\=head(acks))^nnot(acks\=emptyseq)),
                         (A ^ B ^ C))
            = T
```

where `A`, `B`, `C` are respectively the first, second and third subterm of the term Ib:

```
A:[inseq(yvar,ackr) \or inseq(yvar,acks)] \=> [(yvar <= kr)^(ks <= (yvar +1))]
B:[inseq(xrec,cr) \or inseq(xrec,cs)] \=> [(kr <= ind(xrec))^(ind(xrec) <= ks)]
C:[(kr <= ks) ^ (ks <= kr+1)]
```

Proof of (1.1): LP automatically discharges this subgoal using the definition of `init` which represents the initial conditions. It uses the axioms which define the boolean structures and our lemmas about the sequences as `(inseq(exp1,seq1) \=> nnot(seq1\=emptyseq)) = T` to reduce the subgoal to `T=T`.

Proof of (1.2): LP applies the definition of the substitutions, replacing `ks` by `ks+1`, `acks` by `tail(acks)` and then normalizes the goal using the definitions of `<=,+`...:

```
(A ^ B ^ C) \=> [(((ks \= head(acks)) ^ nnot(acks \= emptyseq))
                  \=> ([inseq(yvar, ackr) \or inseq(yvar, tail(acks))]
                          \=> [(yvar <= kr) ^ (ks <= yvar)]
                       ^ [inseq(xrec, cr) \or inseq(xrec, cs)]
                          \=> [(kr <= ind(xrec)) ^ (ind(xrec) <= (ks+1))]
                       ^ [(kr <= (ks+1)) ^ (ks <= kr)]))
                ^ ((nnot(ks \= head(acks)) \or nnot(acks \= emptyseq))
                   \=> (A ^ B ^ C))]
 = T
```

The goal is then normalized to its "normal conjunctive form" using the axiomatization of boolean structures. We supply the theorem prover with the proof method `resume by /\-m`. LP simplifies (or proves) the subgoals using the axiomatization of the transitivity and the appropriate lemmas of the sequences as:

```
(inseq(exp1,tail(seq)) \=> inseq(exp1,seq)) = T
```

which are previously proved. For example, a conjunct of the form:

```
A \or nnot(kr <= ks) \or (kr <= ks + 1) \or B
```

is normalized to `T` using the axiomatization of the naturals and the transitivity.

Nevertheless, LP cannot achieve the proof and stops on the following subgoal:

```
[(inseq(yvar,ackr) \or inseq(yvar,tail(acks)))^(yvar <= kr)^(ks <= yvar + 1)
 ^ (kr <= ind(xrec)) ^ (ind(xrec) <= ks)^(kr <= ks)^(ks <= kr+1)
 ^ (ks \= head(acks)) ^ nnot(acks \= emptyseq)]
 \=> (ks <= yvar)
```

To illustrate the problem, we consider a state s_k (Figure 2) in which the number of the last message sent equals the number of the last message received, $ks = kr = N+1$ and the acknowledgment of this message

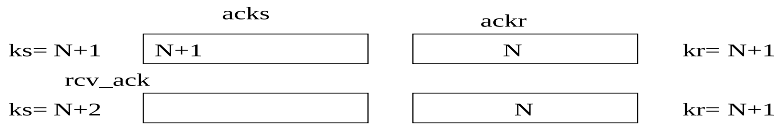

Figure 2 Unreachable state

is present in the head of the acknowledgment channel $head(acks) = N + 1$. Moreover, an old acknowledgment is still present in the acknowledgment channel $ackr$. Clearly, the property I_1 holds in this state. As $ks = head(acks)$ and $acks \neq \emptyset$, action *rcv_ack* is enabled. The corresponding transition results in a state s_{k+1} which verifies: $ks = N + 2$, $kr = N + 1$, $kr \leq ks \leq (kr + 1)$ and $\langle \exists y \in ackr :: not(ks \leq (y + 1)) \rangle$. Property I_1 does not hold in the state s_{k+1}.

On the basis of an intuitive understanding of the behavior of the protocol, we know that the state s_k is **unreachable**, with respect to the initial state. Thus this should not happen according to the program. Indeed we need informations saying that acknowledgments in *acks* are smaller than those in *ackr*, and that *acks* and *ackr* are sequences of nondecreasing integers which is precisely invariant 3.

To prove invariant 3, we must prove that the program *protocol* verifies the property I_3:

invariant *cs.dex, cr.dex, acks, ackr are sequences of nondecreasing integers*

For doing this, the following theorem must be proved in LP for each sequence *acks*, *ackr*, *cs* and *cr*:

```
Inv(issorted(seq),proto,init) = true
```

The predicate `issorted(seq)` is defined† as follow:

```
issorted(emptyseq)             = T
issorted(emptyseq +- exp)      = T
issorted(exp -+ emptyseq)      = T
issorted((seq +- exp1) +- exp2) = issorted(seq +- exp1) ^ (exp1 <= exp2)
issorted(exp1 -+ (exp2 -+ seq)) = issorted(exp2 -+ seq) ^ (exp1 <= exp2)
```

For the sequences *acks* and *cr*, only the actions *rcv_ack*, *receive*, *rmv_old_ack* and *rmv_old_msg* are relevant: the removal of the head do not modify the ordering in the sequence, which we prove by induction by typing the following commands:

```
prove issorted(s) |=>  issorted(tail(s)) = T  by ind depth 2 using seq_ind2
qed
```

This command directs LP to prove the formula `F(s)` by 2-level structural induction. LP generates and automatically discharges the appropriate subgoals using the axioms which define `issorted`; Proving a formula `F(s)` by depth 2 induction using the induction rule

---

†The transmit channels and the acknowledgment channels are formalized as double ended queue, with the two operators $-+$ and $+-$.

`seq_ind2: sort Seq generated freely by emptyseq,-+`
involves proving two basis subgoals, `F(emptyseq)` and `F(emptyseq -+ e)`, and one induction subgoal, `F(e -+ (e1 -+ sc))`, given `F(sc)` and `F(e -+ sc)` as induction hypotheses.

Using this lemma, the following invariants are proved automatically with LP:

```
Inv(issorted(acks),proto,init) = true
Inv(issorted(cr),proto,init) = true
```

For the channels cs and $ackr$, LP can not achieve the proof of the above invariant. The proved lemma is useless as the actions *transmit* and *send_ack* add a new item to the sequence. Taking the hypothesis `issorted(ackr)`, the goal `issorted(ackr +- kr)` can not be deduced. Indeed, this goal is reduced to `last(ackr) <= kr`. As $last(ackr) \in ackr$, using invariant 1, we can deduce $last(ackr) \leq kr$. In the same way, to prove that $cs.dex$ is a sequence of nondecreasing integers, the goal is reduced to `ind(last(cs)) <= ks`. As $last(cs) \in cs$, using invariant 1, we can deduce $last(cs).dex \leq ks$.

Conclusion: the invariance of I_1 requires this of I_3 and vice versa.

We first strengthen the property I_1 with the missing properties: The acknowledgments in $acks$ are smaller than those in $ackr$, $(\forall y \in acks) \wedge (\forall z \in ackr) :: y \leq z$ and with an equivalent property for the transmit channels: $(\forall y \in cs) \wedge (\forall z \in cr) :: z.dex \leq y.dex$. These properties restrict the total state space of the program to the reachable states from the initial state. We obtain the following property:

$$
\begin{array}{ll}
\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\
\langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle & \wedge \\
kr \leq ks \leq kr+1 & \wedge \\
\langle \forall y, z : y \in acks \wedge z \in ackr :: y \leq z \rangle & \wedge \\
\langle \forall x, t : x \in cs \wedge t \in cr :: t.dex \leq x.dex \rangle &
\end{array}
\qquad (I_{aux})
$$

Finally, we prove that the program *protocol* satisfies the property:

Invariant $I_{aux} \wedge I_3$

Actually, the safety proofs are semi-automated. Indeed, to prove a safety property, we use a fixed number of LP commands. The use of LP saves us from supplying the tedious details of the hand proofs. Precisely, arithmetic and logic parts of the proofs are most of the time carried out by the prover. This enables us to focus our study on the given problem. The flaws detected in this specification could not be found by LP, since it is not designed to find difficult proofs automatically. For example, LP does not use heuristics to formulate additional conjectures that might be useful in a proof. Instead, LP makes it easy for us to employ standard techniques such as proofs by cases, induction, and contradiction, either to construct proofs or to understand why they have failed.

In the specification of the first invariant, the predicate Ib is *always-true* , e.g. transitions from a state in which Ib holds to a state in which Ib does not hold are possible, whereas for an invariant such transitions are not possible. Indeed, a predicate is *always-true* in the program if and only if it holds in every initial state and it is maintained by every action when the action is executed in a reachable state. In our example, s_k is not a reachable state. Such a gap was difficult to detect in the corresponding hand proof, due to the sheer amount of details generated by the number of proof obligations (one for each action of the program) and the size and complexity of the predicates. Nevertheless, the hand proof help us to find which lemma to introduce in the mechanized proof and suggests which fact might be relevant to apply.

5.2 The proof of liveness

Proofs of liveness are somewhat different from those of safety. To prove *unless* or *ensures* properties, we apply definitions, it is not the case for the proof of a *leads_to*. This is due to the fact that the first two have an "operational" definition and the last one does not (see the appendix). When we deal with a progress property as p **leads_to** q, we have to check if it is "static" in order to prove p **ensures** q and then infer p **leads_to** q. Otherwise, we must find "good intermediate fact" in order to use the rules which specify the *leads_to* predicate. To avoid this difficulty, we proceed by decomposition to simplify the conjecture. We exhibit intermediate properties to be proved and rules to prove them. One of the most often used rule to prove *leads_to* properties is the *cancellation rule*:

$$\textbf{CAN\_rule}: \quad \frac{leads\_to(p, q \vee b), leads\_to(b, r)}{leads\_to(p, q \vee r)}$$

In the following, we describe the proof of the progress property that the program *proto* must satisfy:

$$|mr| = n \quad leads\_to \quad |mr| = n + 1 \qquad (P)$$

The proof proposed by Chandy and Misra is the following, using kr for $|mr|$, from invariant 2 ($\mapsto$ stands for leads_to):

$$\begin{array}{llr}
kr = n & \mapsto (kr = n+1) \vee (n \in acks) & (P_1) \\
n \in acks & \mapsto ks = n+1 & (P_2) \\
kr = n & \mapsto (kr = n+1) \vee (ks = n+1) & (P_3) \\
 & \qquad \text{CAN_leads_to on } (P_1) \text{ and } (P_2) & \\
ks = n+1 & \mapsto kr = n+1 & (P_4) \\
kr = n & \mapsto kr = n+1 & (P) \\
 & \qquad \text{CAN_leads_to on } (P_3) \text{ and } (P_4) &
\end{array}$$

We had to complete the proof given by the authors, and in particular we expand the proof of (P_2) and (P_4), as we are going to see.

Proof of (P_1): This property is proved using the *conditional property* (COND_prop) of a faulty channel [Chandy and Misra, 1988]. In particular, we consider the faulty channel for the acknowledgments (transmission from the receiver to the sender), taking $p.m \equiv (kr = m)$. The proof consists in a set of *unless* properties.

Proof of (P_2): We first exhibit the fact that when an item n (the acknowledgment of the n*th* message) is present in the acknowledgment channel $acks$, the receive acknowledgment action of the sender ensures that sooner or later this item gets to the head of $acks$ or ks equals $n + 1$. Remember that the sender increments ks when he receives the acknowledgement for the n*th* message. It is exactly the meaning of the induction principle (*see appendix*), using the position of the item in the sequence to be the metric: From any program state in which $(n \in acks)$ holds, the program execution eventually reaches a state in which $(n \in acks \wedge head(acks) = n) \vee ks = n + 1$, or it reaches a state in which $(n \in acks)$ holds and the position of n in the sequence $acks$ is lower (since the position of a item in a sequence cannot decrease indefinitely, $pos(head(seq)) = 1, pos(head(tail(seq))) = 2, \ldots ect)$. Moreover, when an item n is at the head of *acks* we know that ks will be equal to $n+1$ (from the text of *proto*), we can use *ensures* to prove it. Thus the proof of $n \in acks \quad \mapsto \quad ks = n+1$ is decomposed in:

$n \in acks$	$\mapsto ks = n+1 \vee (n \in acks \wedge head(acks) = n)$	IND_leads_to
$n \in acks \wedge head(acks) = n$	$\mapsto ks = n+1$	by ensures
$n \in acks$	$\mapsto ks = n+1$	CAN_leads_to

Proof of (P_4): We have to prove the following property: $ks = n+1 \quad \mapsto \quad kr = n+1$, which asserts that when the sender sends the $(n+1)th$ message, the receiver eventually receives it. The protocol is designed in such a way that if the message is lost the first time it is sent, the sender sends it again unless it receives an acknowledgement. The safety property asserts that any message sent along the channel can be lost, but only a finite number of messages can be lost consecutively.

In order to prove (P_4), we have to decompose the problem, using the conditional property of a faulty channel (COND_prop), which is the only way to prove passing of items between the transmission channel cs and the transmission channel cr.

Proof:

$ks = n+1$	$\mapsto p.m \vee kr = n+1$		(1)
$p.m$	$\mapsto \neg(p.m) \vee (m \in cr)$	COND_prop	(2)
$p.m$	$unless\ (m \in cr) \vee head(cr) = m \vee kr = n+1$		(3)
$p.m$	$\mapsto (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$	PSP_rule(2,3)	(4)
$ks = n+1$	$\mapsto (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$	CAN_leads_to(1,4)	(5)
$p.m \wedge (m \in cr) \wedge pos(m, cr) = i$			
	$\mapsto (p.m \wedge (m \in cr) \wedge pos(m, cr) < i)$		
	$\vee(head(cr) = m \wedge m \in cr) \vee kr = n+1$	by ensures	(6)
$p.m \wedge (m \in cr)$	$\mapsto (head(cr) = m \wedge m \in cr) \vee kr = n+1$	IND_leads_to(6)	(7)
$ks = n+1$	$\mapsto head(cr) = m \vee m \in cr \vee kr = n+1$		
	$\vee(m \in cr \wedge head(cr) = m)$	CAN_leads_to(7,5)	(8)
$head(cr) = m$	$\mapsto m \in cr \wedge head(cr) = m$	IMPL_leads	(9)
$ks = n+1$	$\mapsto kr = n+1 \vee m \in cr \vee (m \in cr \wedge head(cr) = m)$	CAN_leads_to(9,8)	(10)
$m \in cr$	$\mapsto (m \in cr \wedge head(cr) = m)$	IND_leads_to	(11)
$ks = n+1$	$\mapsto kr = n+1 \vee (m \in cr \wedge head(cr) = m)$	CAN_leads_to(10,11)	(12)
$head(cr) = m \wedge m \in cr$	$\mapsto kr = n+1$	by ensures	(13)
$ks = n+1$	$\mapsto kr = n+1$	CAN_leads_to(12,13)	

end proof.

(1) Taking $p.m \equiv (ks = n+1 \wedge ((n+1, ms[n+1]) \in cs))$ and m as $(n+1, ms[n+1])$, we start proving that $ks = n+1$ leads to a situation where the message m is in transit in cs or it has been already received $kr = n+1$.

(2)-(5) Continuing the decomposition, the situation where $p.m$ holds (the message m is in transit in cs) leads to a situation where the message appears in the transmission channel cr.

(6)-(12) Once the transmitted message is in the transmission channel cr, we use the induction principle to prove that it gets to the head.

(13) The receive action of the receiver ensures that sooner or later kr will be equal to the index of m.

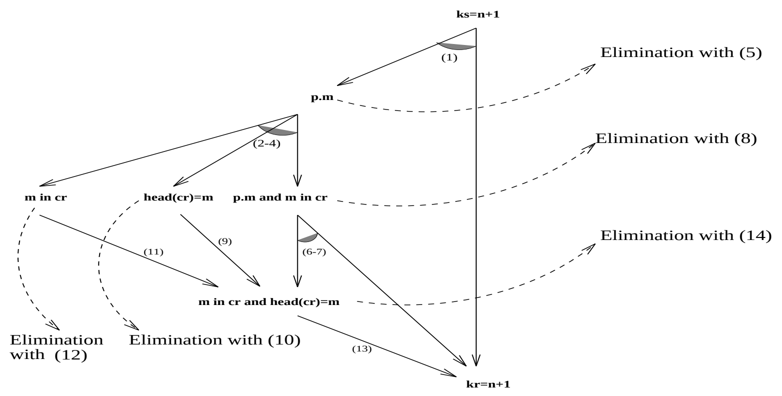

Figure 3 The application of the CAN_rule enables the elimination of the nodes, which represent the intermediate facts. Every triangle represents the application of CAN_rule, and closing the triangle means that the application succeed.

We can represent this proof as a search tree (*proof lattice*), where the root is the predicate which holds in the starting state (Figure 3). Starting from a node, we expand the different possibilities using our intuitive understanding of the behavior of the protocol. Then, we prove smaller properties in order to reduce the tree to a single leaf, which represents the predicate that we want to obtain. The importance of the CAN_rule is that when we want to prove that $p \mapsto q$, we start from p and we try to understand what we want to obtain: whether we have a state where q holds or a state where another predicate r holds. So finding r is crucial, it represents a predicate holding in intermediate state *preceding* the state where q holds, and the CAN_rule allows to get rid of it.
We do not show the details of the LP mechanization for these proofs of liveness, as every proof of a progress property is decomposed to use *ensures* and *unless*. The mechanization of the safety proofs is illustrated by the invariance proofs of the previous section.

6 DISCUSSION AND CONCLUSION

In this paper, we described the verification of a protocol for communications over faulty channels using the UNITY proof theory and its mechanization in LP theorem prover. Indeed, we showed that LP allows the construction of a proof whose structure is identical to that of a careful hand proof. Nevertheless, low steps in hand proofs are usually achieved using intuitive arguments. As those arguments are often based on critical aspects of the problem, they must be formalized and mechanized in order to retrieve errors and mistakes.

Formal reasoning about protocol correctness is necessary to guarantee that protocols work according to their specifications, since informal reasoning is not reliable enough and testing not complete enough. However, such formal reasoning is complex and error-prone task. This complexity is due to the fact that the protocols themselves are complex and exhibit very complicated behaviors. The use of proof tools to check the correctness of the proofs is thus a prerequisite for the success of such a task.

Although the example presented here is a classical one, it is typical of the kinds of proofs that are usually done in distributed systems. In such proofs, we are often faced with a choice between intuitive arguments and careful proofs. The use of a theorem prover forced us to discharge all assumptions based on the intuitive arguments, and then provides a highest degree of confidence in the verification. On another hand, the semi-automatization of the proofs enables us to get rid of the tedious details of the hand proofs, in order to concentrate on the deeper aspects of the problem.

The advantage of this approach is that further optimizations of a verified protocol lead to small modifications in our formalization and in the proofs strategy [Borjesson *et al.*, 1993]. As the protocol is specified as a program and its behavior as program execution, a optimization such as the Alternating Bit Protocol requires only modification of the program actions and does not affect the proof strategy. This is partly due to the fact that most of our safety proofs are purely computational in the sense that they consist of a number of syntactic transformations instead of semantic reasoning steps.

Several works to mechanize verification of UNITY programs through theorem provers have been undertaken, namely with NQTHM [Goldschlag, 1990], and HOL [Andersen, 1992]. The former is an implementation of a mechanized UNITY theory based on a computation model. The NQTHM system uses extensively predefined functions and requires less guidance from the user whereas LP requires to reconstruct all basic functions. But this is an interesting feature since it gives the user more freedom for choosing the functions, for designing the proof and for debugging wrong specifications. The later is a mechanization of UNITY theory in a higher order logic. The reader can find a first comparison with the HOL proofs in [Chetali, 1995b], where we describe the proof of the correctness of a lift-control protocol.

Our future efforts will concentrate on the generalization of the presented approach. In particular we will investigate how we can use it in larger examples, and how redoing proofs in response to changes in the initial specification when scaling the approach to real-size problems.

The main contribution of this paper consists in taking advantage of both the simplicity and the power of the model proposed by Chandy and Misra and those of the assistant prover LP. Distributed systems are difficult in essence. There is no hope to get them right without a formal specification of their behavior and a careful proof of their correctness. We expect we have convinced the reader that we have reached the first aim through UNITY and the second through LP.

REFERENCES

[Andersen, 1992] F. Andersen. *A theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[Borjesson *et al.*, 1993] A. Borjesson, K. M. Larsen, and A. Skou. Generality in design and compositional verification using TAV. *IFIP Transactions C-10: Formal Descriptions Techniques*, pages 449–464, 1993.

[Chandy and Misra, 1988] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988. ISBN 0-201-05866-9.

[Chetali and Lescanne, 1992] B. Chetali and P. Lescanne. An exercise in LP: The proof of the non restoring division circuit. In U. martin and J.M. Wing, editors, *Proceedings First International Workshop on Larch*, volume 780 of *Workshops in Computing*, pages 55–68, Dedham, Boston, August 1992. Springer-Verlag.

[Chetali, 1995a] B. Chetali. Formal verification of concurrent programs: How to specify UNITY using the Larch Prover. Research Report 2475, Inria-Lorraine, 1995.

[Chetali, 1995b] B. Chetali. Formal verification of concurrent programs using the larch prover. In Larsen K.G Engberg, U.H. and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the construction and analysis of Systems*, BRICS Notes, pages 174–186, Aarhus, Denmark, May 1995.

[Dijkstra and Scholten, 1989] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Garland and Guttag, 1991] S. V. Garland and J. V. Guttag. A guide to LP, the larch prover. Technical Report 82, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA., 1991.

[Goldschlag, 1990] D. M. Goldschlag. Mechanically verifying concurrent programs with the boyer-moore prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1022, September 1990.

[Guttag *et al.*, 1993] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Lamport, 1977] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Staunstrup *et al.*, 1988] J. Staunstrup, S. J. Garland, and J. V. Guttag. Verification of VLSI circuits using LP. In *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, pages 329–345. Elsevier Science Publishers B. V. (North-Holland), 1988.

[Staunstrup *et al.*, 1989] J. Staunstrup, S. J. Garland, and John V. Guttag. Localized verification of circuit descriptions. In J. Sifakis, editor, *Proceedings of a Workshop on Automatic Verification Methods for Finite State Systems, Grenoble (France)*, volume 407 of *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, June 1989.

APPENDIX

About LP

The design and development of the Larch proof assistant LP have been motivated primarily to debug LSL specification [Guttag *et al.*, 1993] but it has been also used to establish the correctness of hardware designs [Staunstrup *et al.*, 1989] [Chetali and Lescanne, 1992], or to reason about algorithms involving concurrency [Staunstrup *et al.*, 1988]. It is a general-purpose theorem prover for multisorted first-order logic. It is based on equational term rewriting [Garland and Guttag, 1991], and supports proofs about axiomatic specifications. All proofs are carried out by applying rewrite rules, proofs by case splitting, induction, contradiction and application of inference rules. LP does not contains any predefined theory, but automatically declares the sort *bool* with the corresponding logical operators. A simple specification in LP looks like:

```
declare sorts Elt, Set
 declare variables e,e1: Elt, x: Set
 declare operators
   {}:                      -> Set
   {__}:           Elt      -> Set
   insert:         Elt, Set -> Set
   __ \in __:      Elt, Set -> Bool
   ..
```

Sample axioms for this specification are:

```
set name Axioms
 assert
   sort Set generated by {}, insert;
   {e} = insert(e, {});
   ~(e \in {});
   e \in insert(e1, x) <=> e = e1 \/ e \in x;
   ..
```

The first axiom is an *induction rule*, used in proofs by induction. These axioms (except the first one) are oriented into rewrite rules, for which LP uses the predefined ordering:

```
Axioms.2: {e} -> insert(e, {})
Axioms.3: e \in {} -> false
Axioms.4: e \in insert(e1, x) -> e = e1 \/ e \in x
```

A sample conjecture is: `prove e \in {e}`

The Unity predicates

Unless, *ensures* and *leads_to* are higher-order functions applied to predicates, which are nonmodal formulae. As LP is a first-order language, we had to encode these higher-order functions by first-order functions.

We define a sort Bexp, which is the sort of boolean expressions (boolean structures [Dijkstra and Scholten, 1989]). Two boolean expressions connected by a relational operator, such as $A{=}B$ or $A < B$, is not interpreted as statement of fact but denotes a boolean expression as general as the connected operands. Therefore, as the set Bexp is not (and cannot be) an extension of bool (a basic type provided by LP, we had to define *imply, or, and, true, false* as `\=>`, `\or`, `^`, `T` and `F` different from LP's built-in operators `=>`, , `/\`, `true`, `false` for implication, disjunction and conjunction. So as for `nnot`, `\=` which are different from LP operators `~`, `=`. Then in our system, `A=B` is interpreted as the fact that `A` and `B` are equal "everywhere", and `A \= B` stands for a boolean expression. Moreover, expressions in program variables have types, there are *boolean* expressions or *integer* expressions. Integer expressions are the arithmetic expression built with the operators `+`, `-` , and `*`.

Proofs in UNITY are based on assertions of type $\{P\}s\{Q\}$, equivalent to $P \Rightarrow wp(s,Q)$, where $wp(s,Q)$ is the weakest pre-condition for the post-condition Q [Dijkstra, 1976][‡]. We formalize the wp-calculus for assignment statements using a function: wp:Act,Bexp→Bexp, where act is the sort of actions. We also define list of actions (alist), pairs of identifiers and expressions `(id,exp)` or `(id, bexp)` and finally sets of pairs (pset) that we think of as being unordered lists of pairs of identifiers and expressions.
Here is the axiomatization of wp:

```
Set name wp
assert
  wp(assg(id1.ex),p) = sub_bexp(i1.ex,p)
  wp(cond_assg(id1.ex,b),p) = (b \=> wp(assg(id1.ex),p)) ^ (nnot(b) \=> p)
  wp(mult_assg(l),p) = sub_bexp(l,p)
  wp(cond_mult_assg(l,b),p) = (b \=> wp(mult_assg(l),p)) ^ (nnot(b) \=> p)
```

The function sub_bexp(list,p), where list is a list of pairs (id,exp), substitutes each occurrence of id by exp in the boolean expression p.

Let us now give the definition of the temporal predicates:

```
Set name unless
assert
  unless(p, q, anil) = true
  unless(p, q, a) = (((p ^ nnot(q)) \=> wp(a, p \or q))=T)
  unless(p, q, cons(a,act_l)) = unless(p, q, a) /\ unless(p, q, act_l)
Set name ensures
assert
  ensures(p, q, anil) = false
  ensures(p, q, pgm) = unless(p, q, pgm) /\ exists_act(p, q, pgm)
Set name exist_act
assert
  exists_act(p, q, anil) = false
  exists_act(p, q, a) = (((p ^ nnot(q)) \=> wp(a, q))=T)
  exists_act(p, q, cons(a,act_l)) = exists_act(p, q, a) \/ exists_act(p, q, act_l)
Set name leads_to
assert
  (1) when ensures(p, q, pgm) yield leads_to(p, q, pgm)
  (2) leads_to(p, r, pgm) /\ leads_to(r, q, pgm)) ⇒ leads_to(p, q, pgm)
  (3) leads_to(p, r, pgm) /\ leads_to(q, c, pgm)) ⇒ leads_to(p \or q, r \or c, pgm)
```

[‡] $\{P\}s\{Q\}$ means that "starting from a state where P holds, the execution of s in that state results in a state where Q holds if s terminates". As UNITY statements are assignments, we assume their termination, i.e. $P \Rightarrow wlp(s,Q)$ is equivalent to $P \Rightarrow wp(s,Q)$.

```
Set name Invariant
assert
Inv(p, pgm, init_cond) = ((init_cond \=> p)=T) /\ unless(p,F,pgm)
```

In these definitions, cons(a,act_l) is the list of the actions of the program. For an atomic statement a, unless(p,q,a) means that if p holds and q does not in a program state, then after executing a either q or p holds; hence, by induction on the number of statement executions, p keeps holding as long as q does not hold.
The function exists_act checks whether there is a in pgm such that $\{p \wedge \neg q\}a\{q\}$ is valid. For a given program pgm, ensures(p,q,pgm) implies that unless(p,q,pgm) holds, and if p holds at any point in the execution of the program then q holds eventually.
invariant(p,pgm,cond_init) states that if p holds at every initial state and p is stable, then p holds at every state during any execution of pgm.
In order to infer *p leads_to q* once we have *p ensures q* in the current system, we use what LP calls a **deduction** rule and which it notes when *hypotheses* yield *conclusions*: the *when* clause contains the hypotheses of the rule and the *yield* clause, the fact inferred when the hypotheses are true. We encode the disjunction rule [Chandy and Misra, 1988] for $n = 2$.

An Induction principle for *leads_to*

The following induction rule has is crucial since it involves the fundamental predicate, *leads_ to*, which usually helps to specify progress properties of programs. In most of the proofs of concurrent programs, in order to prove a progress property, we have to exhibit something which "decreases". We give the rule as it was in [Chandy and Misra, 1988]:

$$\textbf{Induction}: \quad \frac{\forall m : m \in W :: leads\_to(p \wedge M = m, (p \wedge M <_w m) \vee q)}{leads\_to(p, q)}$$

W is a set well-founded under the relation $<_w$ and M is a function (the *metric*) from program states to W. For simplicity, in this rule, M (without its argument) denotes the function value when the program state is understood from the context. The hypothesis of this rule is that from any program state in which p holds, the program execution eventually reaches a state in which q holds, or it reaches a state in which p holds with a lower value of the metric M. Since the metric value cannot decrease forever, eventually a state is reached in which q holds.

We have formalized this principle using a lexicographic ordering on list of arithmetic expression. We define this ordering as follow:

```
lexico(nil,nil)=T
lexico(cons(e1,list1),cons(e2,list2)) = (e1<e2) \or ((e1 \= e2) ^ lexico(list1,list2))
```

where list1 and list2 are lists of expressions, e1 and e2 expressions. We define $<: exp, exp \rightarrow bexp$ as an extension of $<: nat, nat \rightarrow bool$. The induction principle is formalized in LP as:

```
Set name IND_leads_to
assert
leads_to((p ^ equal_exp_list(exp_list1,exp_list2)),(p ^ lexico(exp_list1,exp_list2)) \or q,pgm)
  \=> leads_to(p,q,pgm)
```

where (equal_exp_list(L1,L2)) is a function testing "equality" of the two expression lists L1 and L2.