Formal Verification of Concurrent Programs using the Larch Prover

Boutheina Chetali

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

Abstract. This paper describes, by means of an example, how one may mechanically verify concurrent programs on the theorem prover LP. The chosen specification environment is UNITY, a subset of ordinary temporal logic for specifying and verifying programs. We present the proof of a lift-control program, we explain how we can use the theorem proving methodology to prove safety and liveness, and to get semi-automated proofs.

1 Introduction

UNITY [CM88], a fragment of ordinary temporal logic, is a theory to specify and verify concurrent programs. It provides a formalism to express the relevant properties of a program, an appropriate language to construct well-founded formulae and a proof system to construct proofs. Nevertheless, despite the simplicity of the model, formal reasoning about UNITY program correctness is a complex and error prone task. 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 are expressed, without any mention to execution sequences. This *static* view of the program recall the LARCH style of specification [GHG+93], which emphasizes brevity and clarity rather than executability. Moreover UNITY logic is built from a number of modalities called temporal predicates that allow a concise specification of many interesting aspects of programs. These predicates are applied to nonmodal formulae and logical operations or nesting of modalities are not part of the logic. This is interesting aspect for the encoding of UNITY logic on a first-order theorem prover.

In this paper, by means of an example, we show how we mechanize formal proofs of concurrent programs specified in UNITY and the use of a general-purpose theorem prover for first order logic like LP, to verify safety and liveness properties. Our aim is to show that we can take advantage of both the power and the simplicity of UNITY and LP, in order to get semi-automated proofs of safety properties.

The chosen example is taken from [APP93]. It describes a lift-control program specified in UNITY. Its formal proofs of correctness was made with the HOL-UNITY system, which implements UNITY theory in the higher order theorem prover HOL. This example was interesting for several reasons: First, it is a good archetypical problem, second it allows us fruitful comparisons on the use of a theorem prover in mechanizing the proof of correctness of a concurrent programs described in UNITY. Moreover, this example is well-suited to illustrate our approach to get semi-automated proofs of safety properties.

The paper is organized as follows. We start by a brief description of the mechanized implementation of UNITY in LP. Then we present the lift-control program as described in [APP93], and we give the main steps of the mechanization of the correctness proof. For that we focus on one of its safety properties, we present our informal hand proof and the corresponding automated proof in LP. Finally, we conclude with a discussion about this study and future work.

We suppose little familiarity with LP (as few as possible) and we recommand [GG91] as an introduction. This experiment has been conducted with the release 3.1 of LP.

2 About LP

In this section, we give some hints about LP, in order to introduce the formalism used in the remainder of this paper.

The design and development of the Larch proof assistant LP have been motivated primarily to debug LSL specification [GHG+93] but it has been also used to establish the correctness of hardware designs [SGG89] [CL92], or to reason about algorithms involving concurrency [SGG88].

It is a general-purpose theorem prover for multisorted first-order logic. It is based on equational term rewriting [GG91], 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, y: Set
 declare operators
   {}:                       -> Set
   {__}:            Elt      -> Set
   insert:          Elt, Set -> Set
   __ \union __:    Set, 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;
   e \in (x \union y) <=> e \in x \/ e \in y
   ..
```

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
Axioms.5: e \in (x \union y) -> e \in x \/ e \in y
```

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

3 About the encoding of UNITY in LP

A UNITY program consists of 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. It could be an additional section named **always** section used to define shorthands for complicated expressions used in the program.

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. 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 [Lam77]. 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 [Che95b], with the encoding of the syntax of UNITY, the wp-calculus and the syntax of first order predicates. In the following, we give only the definitions needed in this paper.

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[DS89]). 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 [Dij76][1] 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(i1.ex),p) = sub_bexp(p(i1.ex),p)
    wp(cond_assg(i1.ex,b),p) = (b \=> wp(assg(i1.ex),p)) ^ (nnot(b) \=> p)
    wp(mult_assg(pl),p) = sub_bexp(pl,p)
    wp(cond_mult_assg(pl,b),p) = (b \=> wp(mult_assg(pl),p)) ^ (nnot(b) \=> p)
```

[1] $\{P\}s\{Q\}$ 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 $P \Rightarrow wp(s,Q)$.

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)

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. In the definition of the predicate leads_to [CM88], the disjunction rule is really an infinite set of rules, one for each integer n. Indeed, all of these rules are consequences of special case of the rule for $n = 2$. Furthermore, we encode the disjunction rule 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 [CM88]:

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

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.

The function M and the ordering depend on the example or the program to prove. So M and the corresponding order should be considered as parameters to be instancied. That it is not possible in LP, as in almost all theorem provers. For example, M could be a binary function $M \equiv f(x, y) = (x, y)$, with the lexicographic ordering among pairs of integers, or an unary function $f(x) = x + 2$ with the ordering $<$.

So 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 $<: nat, nat \rightarrow bool$.

In LP, we state the induction principle 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 eq is a function testing "equality" of two expression lists.

With this principle, we prove this following rule :

$$\textbf{IND\_coroll}: \quad \frac{\mathsf{leads\_to(pq(l1,l2), lexico(l1,l2), pgm)}}{\mathsf{leads\_to(T, nnot(p), pgm)}}$$

4 A lift-control program

The aim of this section is to show, by means of an example, the mechanization of safety properties with LP and how we deal with a program manipulating naturals, booleans and abstract data type as arrays. The following example is taken from [APP93], which describes a lift-control program. The proof was made with the HOL-UNITY system, which implements UNITY theory in the higher order theorem prover HOL. Therefore we want to investigate the mechanization of the proof on a first order theorem prover like LP.

The lift program describes a lift that moves between a number of floors to serve requests on these. In [APP93] the bottom and top floors are specified with two constant parameters *min* and *max*, we simplify without loss of generality considering three floors : $min = 0$ and $max = 2$.

Program {Lift}
Declare
  $floor : integer,$
  $up, move, stop, open : bool,$
  $req : array[0..2]\ of\ bool,$
initially
  $floor = 0\ []\ up, move, stop, open = false, true, true, false$
  $req[0], req[1], req[2] = false, false, false$
Always
  $above = \exists i : floor < i \leq 2 \wedge req[i]$
  $below = \exists i : 0 \leq i < floor \wedge req[i]$
  $queueing = above \vee below$
  $goingup = above \wedge (up \vee \neg below)$
  $goingdown = below \wedge (\neg up \vee \neg above)$
  $ready = stop \wedge \neg open \wedge move$
assign

$$\begin{array}{llll}
\{request_act\} & stop, move & := true, false & if\ \neg stop \wedge req[floor] \\
[]\{open_act\} & open, req[floor], move & := true, false, true & if\ stop \wedge \neg open \wedge req[floor] \\
 & & & \quad \wedge \neg(move \wedge queueing) \\
[]\{close_act\} & open & := false & if\ open \\
[]\{req_up\} & stop, floor, up & := false, floor + 1, true & if\ ready \wedge goingup \\
[]\{req_down\} & stop, floor, up & := false, floor - 1, true & if\ ready \wedge goingdown \\
[]\{move_up\} & floor & := floor + 1 & if\ \neg stop \wedge up \wedge \neg req[floor] \\
[]\{move_down\} & floor & := floor - 1 & if\ \neg stop \wedge \neg up \wedge \neg req[floor]
\end{array}$$

end{lift}

The state space of the lift is represented by six variables, $floor$ denotes the current position of the lift, $open$ (resp. $stop$) whether the door is open (resp. stopped) at $floor$, $req[i]$ whether the lift is requested at the floor i, up denotes the current direction of the movement, and $move$ whether moving the lift takes precedence over opening of the doors.

5 Translation in LP

Much of the power of the deductive system of LP relies upon *normalization*, e.g. computation of normal forms by rewriting, then the basis for proofs is a logical system, which consists of signatures, equations, rewrites rules, operators theories, ...etc. In addition, LP provides inference rules and tactics for proving theorems, and requires guidance in the proof strategy, e.g. the chaining of elementary proof tactics, and in the introduction of lemmas if necessary. These lemmas are proved and later used as axioms in proofs. In our proof, the suggestions we made were about the proof methods, or explicit instantiations of variables in lemmas, equations, ...etc. In the following, we explain the main steps in the translation of the UNITY program $lift$ in LP.

5.1 State variable and Proof variable

We consider two kinds of UNITY variables. 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, are *static* variables encoded as LP variables of sort Id_of_var[2].

The variables $floor$, up, $move$, $stop$, and $open$ are declared in LP as unary operators of sort Id. The array req is defined as an array of Id.

[2] 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 [Che95b]

5.2 The program

A Unity program is defined as a list of actions and will be encodded in Lp as a constant of sort Actlist: lift : $\rightarrow$ Actlist.

A boolean expression init represents the **Initially** section of the program *lift*:

```
assert
init =
 (id_to_exp(floor) |= nat_to_exp(0))
 ^ bool_to_bexp(id_to_bexp(up) = F) ^ bool_to_bexp(id_to_bexp(move) = T)
 ^ bool_to_bexp(id_to_bexp(stop) = T) ^ bool_to_bexp(id_to_bexp(open) = F)
 ^ bool_to_bexp(id_to_bexp(req[0]) = F)^ bool_to_bexp(id_to_bexp(req[1]) = F)
 ^ bool_to_bexp(id_to_bexp(req[2]) = F)
```

The term `bool_to_bexp(id_to_bexp(up) = F)` represents the initial condition $up = false$, the function id_to_bexp embeds the sort Id in the sort bexp and bool_to_bexp transforms a boolean into a boolean expression.

Identifiers and naturals are themselves expressions. As the logic of Lp does not has subsorts, we explicitly embedded Id, Id_of_var and nat into exp using functions id_to_exp and nat_to_exp.
The **always** section is translated as a set of axioms:

```
set name always
assert
queueing = (above \/ below);
goup     = (above /\ ((id_to_bexp(up)=T) \/  ~(below)));
godown   = (below /\ ((nnot(id_to_bexp(up))=T) \/ ~(above)));
ready    = ((id_to_bexp(stop) ^ (nnot(id_to_bexp(open)) ^ id_to_bexp(move)))=T);
above    = (\E ex_i ((((id_to_exp(floor)<id_to_exp(ex_i))
                     ^(id_to_exp(ex_i)<=nat_to_exp(2)))
                     ^(id_to_bexp(req[id_to_nat(ex_i)])))=T));

below    = (\E ex_j ((((nat_to_exp(0)<=id_to_exp(ex_j))
                     ^(id_to_exp(ex_j)<id_to_exp(floor)))
                     ^id_to_bexp(req[id_to_nat(ex_j)]))=T))
..
```

The **assign** section is encoded as a list of actions:
lift = cons(*request_act*, cons(*open_act*, ..., cons(*move_down_act*, nil)))

For example, the action *request_act* is translated in Lp using the function cond_mult_assg(plist, bexp) which encodes conditional multiple assignments:

```
cond_mult_assg({stop.T} \u ({move.F}\u {}),
    (nnot(id_to_bexp(stop)) ^ (id_to_bexp(req[id_to_nat(floor)]))))
```

6 Proof Obligations

In this section, we present the proof as it is described in [APP93], in order to show its mechanization in our system.

The proof obligations describe properties that the *lift* program must satisfy. For that, the main property to prove is that any request for service will eventually be served:

$$\forall n : min \leq n \leq max \Rightarrow req[n] \ \textbf{leads\_to} \ \ open \wedge floor = n \qquad (1)$$

As in HOL-Unity, a property *leads_to* must be satisfied for any state in the total state space spanned by the program variables. Hence, a predicate *valid* is introduced, which characterizes a subset of the total state space including the reachable states of the program $lift$:

$$\forall n : min \leq n \leq max \Rightarrow req[n] \wedge valid \ \textbf{leads\_to} \ open \wedge floor = n \qquad (2)$$

This property is decomposed into simpler UNITY properties [APP93], from which it can be deduced using *leads_ to* inference rules.

Note: In the following, $\mapsto$ stands for *leads_ to*, and p **op** q for $\textbf{op}(p, q, lift)$. We use s for the variable $stop$, o for $open$, m for $move$, f for $floor$, q for $queueing$, a for $above$, b for $below$, $goup$ for $goingup$.

The decomposition of (2) leads to the following properties, the proof of each one follows from the corresponding *ensures* properties:

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$
$$s \wedge \neg o \wedge \neg m \wedge (f \neq n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f \neq n) \wedge req[n] \qquad (P_2)$$
$$s \wedge o \wedge m \wedge (f \neq n) \wedge req[n] \wedge valid \mapsto s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req[n] \qquad (P_3)$$
$$s \wedge o \wedge m \wedge (f = n \wedge \neg q) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_4)$$
$$s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req[n] \wedge valid \mapsto \neg s \wedge \neg o \qquad (P_5)$$
$$\neg s \wedge \neg o \wedge req[n] \wedge valid \mapsto s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \qquad (P_6)$$

To illustrate the next section, let us explain the main steps of the proof of P_1, which states that if the lift is moving and there is a request at the floor n then the lift will stop at that floor and the door will open:

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$

$\Leftarrow$*{Definition of leads_ to}*

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \ \textbf{ensures} \ s \wedge o \wedge m \wedge (f = n) \wedge req[n] \qquad (P_1)$$

$\Leftarrow$*{Definition of ensures}*

$$[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \ \textbf{unless} \ s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \qquad (P_{1.1})$$
$$\wedge[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid \ \textbf{exit\_act} \ s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \ (P_{1.2})$$

$\Leftarrow$*{Definition of unless}*

$$\begin{aligned}
[\forall act \in lift \ [(&s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\Rightarrow \textbf{wp}(act, (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\qquad \vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]] \qquad (P_{1.1}) \\
\wedge[s \wedge \neg o \wedge \neg m \wedge (f = n) &\wedge req[n] \wedge valid \ \textbf{exit_act} \ s \wedge o \wedge m \wedge (f = n) \wedge req[n]] \ (P_{1.2})
\end{aligned}$$

Proof of $(P_{1.1})$: This subgoal leads to six subsubgoals, each per action of the program. Therefore, in order to keep the length of the proof within a reasonable size, we present the main steps using the first action of the $lift$ program.

$$\begin{aligned}
[(&s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\Rightarrow \textbf{wp}(req_act, (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\qquad \vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]
\end{aligned}$$

{Definition of wp and assignments}

$$\begin{aligned}
(&s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid) \\
&\Rightarrow [((\neg s \wedge req[f]) \Rightarrow \ \textbf{sub_bexp}(\{(s.true), (m, false)\}, \\
&\qquad\qquad (s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)
\end{aligned}$$

$$\vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n])))$$
$$\wedge(\neg(\neg s \wedge req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n]))))]$$

{Definition of substitution}

$$(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow [((\neg s \wedge req[f]) \Rightarrow ((true \wedge \neg o \wedge \neg false \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\vee(true \wedge o \wedge false \wedge (f = n) \wedge req[n])))$$
$$\wedge(\neg(\neg s \wedge req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n]))))]$$

{Definitions of $\wedge$,$\vee$ and booleans }

$$(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow [((\neg s \wedge req[f]) \Rightarrow (\neg o \wedge (f = n) \wedge req[n] \wedge valid))$$
$$\wedge((s \vee \neg req[f]) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\vee(s \wedge o \wedge m \wedge (f = n) \wedge req[n]))))]$$

$[true]$ □

Proof of $P_{1.2}$:

$[s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid$ **exit_act** $s \wedge o \wedge m \wedge (f = n) \wedge req[n]]$

{Definition of exis_act}

$$[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow \mathbf{wp}(req\_act, (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]$$
$$\vee[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow \mathbf{wp}(open\_act, (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]$$
$$\vee \ldots$$
$$\vee[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow \mathbf{wp}(move\_down\_act, (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]$$

{only open_act verifies the *wp* property}

$$[false]$$
$$\vee[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow [((s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge true \wedge true \wedge (f = n) \wedge req[n]))$$
$$\wedge(\neg(s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]]$$
$$\vee[false]$$

{Simplifications}

$$[false]$$
$$\vee[(s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req[n] \wedge valid)$$
$$\Rightarrow [((s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge (f = n) \wedge req[n]))$$
$$\wedge(\neg(s \wedge \neg o \wedge req[f] \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge (f = n) \wedge req[n]))]]$$
$$\vee[false]$$

{Using $((f = n) \wedge req[n] \wedge \neg q) \equiv false)$}

$[true]$ □

In the next section, we illustrate the mechanized proof by showing how LP is used to prove the progress property P_1. Precisely, LP automatically discharges the appropriate subgoals in the informal proof and saves us from supplying the tedious details in manipulating the logic parts of the proof goals.

7 The mechanized Proof

In LP, to prove that *p unless q* holds for the lift program, we must prove that the term `unless(p,q,lift)` boils down to `true`.

We introduce the conjecture typing:

```
prove unless(p,q,lift)
```

Using the definition of `unless`, LP expands the conjecture in:

```
unless(p, q, request_act) /\ unless(p, q, cons(open_act,cons(..,nil))).
```

Each conjunct is expanded, introducing `wp`, the normalization goes on and leads to the term[3]:

```
((p ^ nnot(q)) \=> wp(request_act,p \or q))
  /\ ((p ^ nnot(q)) \=> wp(open_act,p \or q))
   /\ ...
      /\ ((p ^ nnot(q)) \=> wp(move_down_act,p \or q))
```

Following the informal hand proof of the previous section, we must divide this conjecture in six subgoals, one per action. We supply LP with two forms of user guidance:

The command `resume by /\-method` directs LP to prove each of the subgoal of the current conjecture as a separate subgoal. It generates the appropriate subgoals of the form

```
((p ^ nnot(q))=T) => (wp(act,p \or q)=T).
```

For each subgoal, the command `resume by =>-m` directs LP to use the implication method. It generates the hypothesis `((p ^ nnot(q))=T)` and automatically discharges the subgoal `wp(act,p \or q)=T)`. To complete the proofs, it uses the axioms and rewrite rules which define the `^`, `\or` and properties of the boolean expressions.

In this proof, we made no mention to the formulae `p` or `q`. Actually, what appears in the following script is the entire interaction with LP for the proof of the *unless* part of the property $P_{1.1}$:

```
set name property1
prove
  unless(stopped^(id_to_exp(floor)|=id_to_exp(nn))
          ^id_to_bexp(req[nn])
          ^(id_to_exp(nn)<=nat_to_exp(2))^(nat_to_exp(0)<=id_to_exp(nn)),
         opened^(id_to_exp(floor)|=id_to_exp(nn))
          ^id_to_bexp(req[nn])
          ^(id_to_exp(nn)<=nat_to_exp(2))^(nat_to_exp(0)<=id_to_exp(nn)),
         lift)
..
   resume by /\-method
```

[3] Actually, we do not write the normal form of the subterms `wp(..)`

```
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
       resume by =>-method
qed
```

To prove the properties P_2,P_3, and P_4 we use the same script[4] than the one of P_1. To make the proving of the property P_5 easier, we first prove an additional property:

`(~above /\ ~below /\ (req[n]=T) /\ (0<=n<=2))  => (floor = n)`

which states that "if there is no request at the floors upper than the *floor* where the lift is currently located, nor at the floors lower than *floor* and there is a request at the floor n then $floor = n$". The predicate *valid* is proved as an invariant in our system and implicitly added to each property to prove.

More technically The general tactic is the following: Let k be the number of statements in a given program *pgm*. For each proof of an *unless* property, we use one command `resume by /\-m` and k commands `resume by =>-m`. The proofs of an *invariant* use one command `resume by /\-m` and $k+1$ commands `resume by =>-m`. The additional subgoal is `(init \=> I)` given by the definition of `invariant(p,init,pgm)`.

The proofs of the progress property *ensures* involve a `resume by /\-method` which leads to a number of subgoals equal to $(k+1)$, where k is as above the number of actions in the program. The additional subgoal corresponds to the *exist_act* part in the definition of `ensures(p,q,pgm)`. This subgoal is proved by cases, `(p ^ nnot(q)) = T` and `(p^nnot(q)) = F`. In order to derive the corresponding *leads_to* properties, Lp uses the deduction rule (1) in the definition of the *leads_to* predicate.

Note that we can define a general proof strategy using the command:

`set proof-methods normalization, /\, =>`

This commands directs Lp to attempt to prove conjectures by the first applicable proof method specified in the given list. So to prove an *unless* property, Lp uses the `/\` method to split the conjecture after rewriting it to its normal form, and uses the `=>` method for each subgoal. As the proof obligations of p *unless* q or *invariant* p are the same for all p and q, the mechanized proof could be fully automated with a single command and the corresponding script will be

```
prove unless(p,q,pgm)
qed
```

Although this strategy leads to less interaction with Lp, we do not use it in order to keep control on the lower steps of the proof. Indeed, as our goal is primarily to check proofs and safety properties, the full mechanization of a proof leads us, as often in such an experiment, to enter low-level detail of the specification. In most of the cases, the low-level steps carry the deeper aspect of the problem. 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[Che95a].

[4] The command `qed` requests Lp to confirm that the proof is indeed complete.

8 Discussion and Conclusion

In this paper, we have proved the correctness of a lift-control protocol. We have used this example as an illustration of our approach of fully mechanizing the correctness using the theorem prover Lp. This process requires first a specification in Unity. Unlike others, our approach lies on a reasoning on the text of the program which we feel more intuitive and closer to the problems we have to solve. Moreover our mechanized Lp proof follows as closely as possible a hand proof a user could make, adding to the hand proof the scrutinity of the theorem prover which should take out all mistakes of the original specification.

For a first comparison with the HOL proofs, we note that we made no mention to *state* and *execution* in our proofs. In Andersen, Petersen, and Pettersson [APP93] proofs, the variable names, the program constructs and the logical connectors are state lifted. The organization of the proof is also differents. The proof of a property that must be satisfied by the program is derived from smaller proofs, each per action, when our proof deals with the whole program. They obtain proofs of minimum size but a number of proofs linear in the number of program actions. In our proofs have exactly the opposite behaviour, e.g the number of commands used in each proof is linear in the number of program actions. This approach allows to handle larger examples, since the proofs of safety properties have the same architecture, e.g the number of commands used depends on the number of actions in the given program.

The whole tactics used for the progress property *leads_ to* in both proofs are similar. In both formalizations, the proof of a *leads_ to* requires a decomposition of the given property into simpler ones in order to use the derived inference rules. This decomposition is based most of the time on intuitive arguments and the use of the *cancellation rule* of the predicate *leads_ to* [Che95a].

The extra amount of time needed to prove the arithmetic parts of each subgoal could be advocated against our approach. But these additional lemmas could be proved and put in a separate library and credit should be given to HOL for its rich arithmetic library.

Our implementation of Unity contains all definitions, theorems, corollaries defined in [CM88], but the axiom of substitution has been removed in order to preserve soundness [B.A91]. All the theorems (derived rules) about *unless* and *ensures* are proved in Lp [Che95b], whereas the rules about *leads_ to* are not since they require structural induction on the length of proof that is not available in Lp.

Our future efforts will concentrate on the generalization of the presented approach and the design of a translator, which will translate automatically a Unity program in a script Lp, and proof obligations as input to Lp.

References

[APP93] F. Andersen, K.D. Petersen, and J.S. Pettersson. Program verification using HOL-Unity. In J.J. Joyce and C.H. Seger, editors, *Proceedings sixth International Workshop on Higher Order Logic theorem proving and its applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–15, Vancouver, Canada, August 1993. Springer-Verlag.

[B.A91] Sanders B.A. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[Che95a] B. Chetali. A formal proof of a protocol for communications over faulty channels using the Larch Prover. Research Report 2476, Inria-Lorraine, 1995. submitted.

[Che95b] B. Chetali. Formal verification of concurrent programs: How to specify UNITY using the Larch Prover. Research Report 2475, Inria-Lorraine, 1995.

[CL92] B. Chetali and P. Lescanne. An exercice 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.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Fundation.* Addison-Wesley, 1988. ISBN 0-201-05866-9.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DS89] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[GG91] 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.

[GHG+93] 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.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[SGG88] 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.

[SGG89] 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.

This article was processed using the LaTeX macro package with LLNCS style