Formal Verification of Concurrent programs: How to specify UNITY using the Larch Prover

Boutheina Chetali

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Formal Verification of Concurrent programs: How to specify UNITY using the Larch Prover

Boutheina Chetali

N° 2475

Janvier 1995

PROGRAMME 2

ISSN 0249-6399

Formal Verification of Concurrent programs: How to specify UNITY using the Larch Prover

Boutheina Chetali*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Eureca

Rapport de recherche n2475 — Janvier 1995 — 15 pages

Abstract: This paper describes the use of the Larch Prover to verify concurrent programs. The chosen specification environment is UNITY, because it provides a higher level of abstraction to express solutions to parallel programming problems. We investigate how the syntax and the semantic of UNITY can be mechanized in LP, a theorem prover designed to check and reason about algebraic specifications, and how we can use the theorem proving methodology to prove safety and liveness

Key-words: formal verification, concurrent program, Unity, Larch prover

(Résumé : tsvp)

*CRIN-CNRS and INRIA-lorraine, University of Henri Poincaré, B.P.239,54506 Vandoeuvre-les-Nancy, email: chetali@loria.fr

Unité de recherche INRIA Lorraine
Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Vérification Formelle de programmes Concurrents: Comment spécifier UNITY avec le Larch prouveur

Résumé : Cet article décrit l'utilisation du prouveur de théorèmes LP pour vérifier des programmes concurrents. L'environnement de spécification choisi est UNITY. Ce mod'ele fournit un niveau élevé d'abstraction pour exprimer les solutions aux problèmes de programmation parallèle. Nous avons étudié la formalisation de la syntaxe et de la sémantique de UNITY en LP, prouveur concu à l'origine pour raisonner sur les spécifications algébriques, dans le but d'utilisé ce prouveur pour la preuve de propriété de *sûreté* et de *vivacité*.

Mots-clé : vérification formelle, programme concurrent,Unity, Larch prouveur

1 Introduction

Several approaches to program verification have been proposed and used in literature. Among them, there are two principal views, one operational the other syntactic. The first one consists of an analysis of 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 [CM88a] provides these 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, the use of a theorem prover provides a very high degree of confidence that this verification is correct. UNITY offers a logic and a notation in which abstract specification of the computation and successive refinements of the specification are expressed, without any mention to the execution sequences. This *static* view of the program recall the LARCH style of specification [GHG+93], which emphasizes brevity and clarity rather than executability.

In this paper, our aim is to investigate how UNITY logic and methodology could be specified within a general-purpose theorem prover for first order logic like LP [GG91], and we are interested in using it to verify safety and liveness properties of concurrent programs written in UNITY. Proofs are done in UNITY logic, an extension of the language UNITY.

The paper is organized as follows. We start by an introduction to UNITY, and its fundamentals concepts. Next we describe the mechanization of UNITY logic, give the specifications of temporal operators in the framework of the assistant prover LP, and give an example of proof of one UNITY inference rule. Finally, we conclude with a discussion about this study and future work.

2 A brief introduction to UNITY

A UNITY program consists of declarations of variables, a description of their initial values and, a finite set of statements. All statements in UNITY are assignments, therefore execution of every statement 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.

Proofs in UNITY are based on assertions of type $\{P\}s\{Q\}$, where s represents a UNITY assignment statement, P a precondition that must be true before the execution of s and Q a postcondition that results from an execution of s. This notation denotes that execution of statement s in any state that satisfies predicate P results in a state that satisfies predicate Q, if execution of s terminates. This assumes implicitly that execution of every UNITY atomic statement (assignment) terminates.

The temporal operators

Unless For a given program, p *unless* q denotes that once predicate p is true, it remains true at least as long as q is not true. Formally:

$$p \textbf{ unless } q \stackrel{def}{=} \langle \forall s : s \ in \ pgm \ :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

i.e, if $p \wedge \neg q$ holds prior to execution of any statement of the program, then $p \vee q$ holds following the execution of that statement.

Ensures For a given program, p *ensures* q implies that p *unless* q holds for the program, and if p holds at any point in the execution of the program then q holds eventually. Formally:

$$p \textbf{ ensures } q \stackrel{def}{=} p \ unless \ q \ \& \ \langle \exists s : s \ in \ pgm \ :: \{p \wedge \neg q\} s \{q\} \rangle$$

It follows from this definition that once p is true it remains true at least as long as q is not true. Furthermore, from the rules of program execution, statement s will be executed some time after p becomes true. If q is still false prior to the execution of s, then $p \wedge \neg q$ holds, and the execution of s establishes q.

Leads_to For a given program, p *leads_to* q denotes that once p is true, q is or becomes true. Unlike *ensures*, p may not remain true until q becomes true. **leads_to** is defined as the strongest predicate satisfying:

$$\frac{p \ \ ensures \ \ q}{p \ leads\_to \ q}$$

$$\frac{p \ \ leads\_to \ \ r, r \ \ leads\_to \ \ q}{p \ \ leads\_to \ \ q}$$

For any set W,

$$\frac{\langle \forall m : m \in W :: p(m) \ \ leads\_to \ \ q \rangle}{\exists m : m \in W :: p(m) \rangle \ \ leads\_to \ \ q}$$

3 Encoding UNITY using LP

The logic of the theorem prover in which it is intended to mechanize another notation must be both powerful and general enough to allow a sound and practical formulation. Several theorem provers have been used to mechanically verify UNITY programs, namely NQTHM [Gol90], B_TOOLS [BM93], HOL [And92].

Unlike the NQTHM system or B , the Larch Prover does not contain any predefined theory. LP is based on equational term rewriting [GG91], for a fragment of first-order logic, 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. 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], and to reason about algorithms involving concurrency [SGG88].
Our purpose in stating this work was to check whether the logic supported by LP can be used to encode the logic of UNITY, and if the style of LP specification could be used to specify a UNITY program. We apply this to verify properties about concurrent programs. That goal requires to encode the syntax of UNITY, the wp-calculus and the syntax of first order predicates.
We suppose a little familiarity with the LP system (as few as possible) and we recommand [GG91] as an introduction.

3.1 Sorts and Types

The model proposed by UNITY is traditional in the sense that we have 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.

The first section in a UNITY program, declare, contains the variable names and their types, and the section initially defines their initial values.

Example: **declare**
$kr : nat$
$bb : bool$
initially
$kr = 1, bb = false$

In this declaration, kr and bb are variable names. In a *state-transition* model, the values of theses variables represent at any moment of the computation a program state. They take their values from different types. But we can have to prove a property like : $\forall y : y \leq kr \leq (y+1)$. In this property, kr is a identifier of type nat and y is a identifier of variable of type nat.

Therefore UNITY variables are of two kinds. The identifiers appear in the program text and 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.

It is important to distinguish between LP variables, logical variables which are identifier standing for an arbitrary value of some sort, and UNITY program variables (*state* variables). Logical variables do not change over time.
The sorts used are: naturals (nat), sequences (seq), record (rec), and boolean. We also define arithmetic expressions (exp), boolean expressions (bexp) such as $((x + y) > 0) \wedge b$, actions and list of actions (act, 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.

Specifications of the abstract data types (naturals, sequences,...etc) are part of the Larch Shared Language [GHG+93], which we have modified in order to axiomatize customized data types. These specifications are definitional in the sense that they list explicitly the properties of the type, which we can manipulate just invoking its operations. We will see that is not the case for the specification of temporal operators.

Embedding

The operator +: exp,exp → exp creates a new expression out of two expressions. Clearly this operator is not the same as the operator +: nat,nat → nat, but as we may imagine it is strongly connected, and 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.

- nat_to_exp : transforms a natural into an expression.
- id_to_exp: transforms an identifier into an expression.
- id_to_nat: transforms an identifier into an natural.
- id_to_bexp: transforms a boolean identifier into an boolean expression.
- seq_to_exp : transforms a sequence into an expression.

Note: The signature of *id_to_nat* is: id → nat and id_of_var → nat. For instance, kr is a natural identifier of the sort id, x is a variable identifier of the sort id_of_var and we could have $id\_to\_nat(kr) = id\_to\_nat(x)$.

3.2 Assignment

We define a UNITY program as a list of actions and we assign this list to a constant: prg : → actlist.
To encode assignment, we use four operators:

- assg: pair → act: single assignment with pair of type (id, exp) or $(id, bexp)$.

- cond_assg: pair, bexp → act: conditional assignment where *bexp* is the condition.
- mult_assg: pset → act: multiple assignment, where *pset* is a list of pairs (id, exp).
- cond_mult_assg: pset, bexp → act: multiple conditional assignment, where *pset* is a list of pairs and *bexp* the condition.

A multiple assignment is a parallel assignment, but we cannot directly code the fact that all assignments in the list are carried out simultaneously. In the logic of LP, the corresponding rule would be an infinite rewrite rule. Thus we use the fact that LP allows associative and commutative operators, and *pset* is a set of pairs with the union declared to be associative and commutative. We obtain then a unordered list of pairs of identifier and expressions This representation has the property that a variable may occur more than once in the left-hand side of an assignment. It is the programmer's responsibility to ensure for any such variable that all possible values that may be assigned to it in a statement are identical [CM88a].

3.3 The wp_calculus

The assertion $\{P\}s\{Q\}$ is a notation equivalent to $P \Rightarrow wp(s, Q)$, where $wp(s, Q)$ is the weakest pre-condition for the post-condition Q [Dij76]. We have formalized the wp-calculus for assignment statements. Let $X := \Sigma \; if \; b$, be an assignment statement where X is a list of identifiers, Σ a list of expressions possibly depending on X, and b a boolean expression. The semantic of this assignment is:

$$\begin{array}{lcl} wp(x := E, Q) & \equiv & Q_x^E \\ wp(x := E \; if \; b, Q) & \equiv & (b \Rightarrow wp(x := E, Q)) \wedge (\neg b \Rightarrow Q) \\ wp(X := \Sigma \; if \; b, Q) & \equiv & (b \Rightarrow Q_X^\Sigma) \wedge (\neg b \Rightarrow Q) \end{array}$$

where Q_x^E is obtained by substitution in Q of all occurrences of x by E, similarly Q_X^Σ is obtained from Q by simultaneous substitutions of the variables $\{x_1 \ldots x_n\}$ in X by the expressions $\{e_1 \ldots e_n\}$ in Σ.

3.3.1 Specification of assignment

```
Set name wp
assert
    wp(assg(id1.ex),p) == sub_bexp(p(id1.ex),p)
    wp(assg(id1.b),p) == sub_bexp(p(id1.b),p)

    wp(cond_assg(id1.ex,b),p) == (b |=> wp(assg(id1.ex),p)) ∧ (nnot(b) |=> p)
    wp(cond_assg(id1.c,b),p) == (b |=> wp(assg(id1.c),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)

In this specification, the first two lines specify assignment of an expression ex or a boolean expression c to an identifier. 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 |⇛,∨ and ∧ different from LP's built-in operators ⇒ | & for implication, disjunction and conjunction. So as for nnot,\= which are different from LP operators not,=. 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. We specify substitutions inductively on the structure of expressions.

3.4 Specification of the temporal operators

As for the operator wp, the specifications of the temporal operator are *operational* specifications, which define a function by showing how to compute it. Let us give the equational specification of the three logical operators which are at the core of the UNITY logic.

Unless:
unless(p,q,anil) == true
unless(p, q, a) == (((p∧¬(q))|⇛ wp(a, p∨q))=T)
unless(p,q,a@act_l) == unless(p,q,a) & unless(p,q,act_l)

a@act_l is the list of all 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.

Ensures:
ensures(p,q,anil) == false
ensures(p,q,pgm) == unless(p,q,pgm) & exists_act(p,q,pgm)
where
exists_act(p,q,anil) == false
exists_act(p,q,a) == (((p ∧¬(q)) |⇛ wp(a,q))=T)
exists_act(p,q,a@act_l) == exists_act(p,q,a) ∨ exists_act(p,q,act_l)

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.

Leads_to:
when ensures(p,q,pgm) yield leads_to(p,q,pgm)
leads_to(p,r,pgm) ∧ leads_to(r,q,pgm)) ⇒ leads_to(p,q,pgm)
leads_to(p,q,pgm) ∧ leads_to(r,q,pgm)) ⇒ leads_to(p ∨ r,q,pgm)

leads_to(p,r,pgm) ∧ leads_to(q,c,pgm)) ⇒ leads_to(p ∨ q,r ∨ c,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 [CM88b], 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 code the disjunction rule for $n = 2$, and a special case of the general disjunction theorem [CM88b].

Note: We do not code the last one as a deduction rule because such a rule would lead the system into an infinite loop. If we would have p_1 *leads_to* q_1 and p_2 *leads_to* q_2 in our current system, LP would infer $(p_1 \vee p_2)$ *leads_to* $(q_1 \vee q_2)$ and with the last one it would infer $(p_1 \vee (p_1 \vee p_2))$ *leads_to* $(q_1 \vee (q_1 \vee q_2))$ and so on. This is due to the fact that there will be four different variables in the hypotheses of the deduction rule. It is not the case for the second and the fourth axioms because there is the same variable in the hypotheses, but for uniformity we encode the four axioms as implications and we use them with explicit instantiation (*see (4)*).
The next operators are abbreviations.

Invariant:

stable p is a shorthand for p **unless** *false* or in LP language:

stable(p,pgm) == unless(p,F,pgm)

If p holds at every initial state and p is stable, then p holds at every state during any execution of pgm; it is an *invariant* of pgm:

Inv(p,pgm,init) == ((init ⊨ p)=T) & stable(p,pgm)

Fixed point:

A fixed point of a program is defined by :

$$FP \equiv \langle s : s \in pgm \wedge s \ is \ X := E :: X = E \rangle$$

This property is not an operator like *unless*, *ensures* ... etc. It is a boolean expression, which we compute using a recursive function on the actions of the program.

$$
\begin{array}{lcl}
\mathsf{fp(l(assg(i1.e)))} & == & \mathsf{rec(p(i1,e))} \\
\mathsf{fp(l(mult\_assg(p(i1,e)@plist)))} & == & \mathsf{rec(p(i1,e))} \wedge \mathsf{rec(plist)} \\
\mathsf{fp(l(cond\_assg(i1.ex,b)))} & == & (\mathsf{nnot(b)} \vee \mathsf{rec(p(i1.ex))}) \\
\mathsf{fp(l(cond\_mult\_assg(pl,b)))} & == & (\mathsf{nnot(b)} \vee \mathsf{rec(pl)}) \\
\mathsf{fp(a1@act\_l)} & == & \mathsf{fp(a1)} \wedge \mathsf{fp(al)} \\
\mathsf{fp(anil)} & == & \mathsf{T}
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathsf{rec(pnil)} & == & \mathsf{T} \\
\mathsf{rec(p(i1.e)@pl)} & == & (\mathsf{id\_to\_exp(i1)} = \mathsf{e}) \wedge \mathsf{rec(pl)}
\end{array}
$$

In this specification, i1 stands for an identifier (in1,iv1,is1 or ib1), and e for an expression (ex) or a boolean expression (p)

4 Proofs rules

The UNITY proof system is built upon rules derived from the definition of *unless*, *ensures* and *leads_to*. Derived rules play the same role in proofs of program properties as lemmas in mathematical proofs. So we can introduce them as axioms in the proof or to use them as theorems. For the last case, they have to be proved before any attempt to prove a program. Actually we proved them using LP and put them into a data basis of general purpose UNITY lemmas that can be reused for other experiments. Some of those rules are:

$$\textbf{Unless\_cancel} \quad : \frac{unless(p,q,pgm), unless(q,r,pgm)}{unless(p \vee q, r, pgm)}$$

$$\textbf{Ensures\_conjonc} \quad : \frac{ensures(p,q,pgm), ensures(r,c,pgm)}{ensures(p \wedge r, q \vee c, pgm)}$$

$$\textbf{Leads\_to\_cancel} \quad : \frac{leads\_to(p, q \vee b, pgm), leads\_to(b, r, pgm)}{leads\_to(p, q \vee r, pgm)}$$

We prove most of the rules about *unless* and *ensures* by induction on the list of the actions of the program, because these predicates are defined inductively on the actions of the program.

4.1 Example

As an example, we describe the main steps of the proof of the derived rule *unless_cancel* for an action a (let l be an operator of signature $act \rightarrow act\_list$), followed by its mechanic implementation in LP.

$$\textbf{unless\_cancel}: \quad \frac{unless(p,q,l(a)), unless(q,r,l(a))}{unless(p \vee q, r, l(a))}$$

Proof:

$unless(p,q,l(a))$	**hyp** (1)
$unless(q,r,l(a))$	**hyp** (2)
$unless(p \vee q, (\neg p \wedge r) \vee (\neg q \wedge q) \vee (p \wedge r), l(a))$	**unless_disj** on (1) and (2)
$unless(p \vee q, (\neg p \wedge r) \vee (p \wedge r), l(a))$	($\wedge$ *and* $\vee$) properties
$((\neg p \wedge r) \vee (p \wedge r)) \Rightarrow r$	**lemma** (3)
$unless(p \vee q, r, l(a))$	**ConseqWeak** on (1) and (3)

end Proof.

The LP script of the proof is:

```
-LP script ------------------------------------------------------------------------------
set name cancel_act
prove when unless(p,q,l(a1)),  unless(q,r,l(a1))
      yield unless((p \vee q),r,l(a1))

 inst p by pc, q by qc, c by qc, d by rc , a1 by a1c in
      disjunction_act

 set name cancel_lemma
   prove (((nnot(p)^r) \vee (q^r)) \=> r)=T==true by =>-m
     resume by case (qc ^ rc) = T
 inst p by (pc \vee qc), q by ((nnot(pc)^rc) \vee (qc ^rc)),
      r by rc,a1 by a1c in consqweak_act
qed
-----------------------------------------------------------------------------------------
```

The inference rule *unless_cancel* is translated in a deduction rule. To prove a deduction rule, LP transforms the hypotheses by replacing each variable by a constant (p by pc, q by qc and r by rc) and adds this hypothesis to its current system and tries to prove the conclusion. The command inst directs LP to use the inference rule *unless_disj* and *conseqWeak*, two deduction rules which we did not give here but which the system knows, with the appropriate constants. We direct LP to prove the lemma cancel_lemma by $\Rightarrow$ method: First LP normalizes the conjecture, then adds to the current system the hypothesis:

```
 ((nnot(pc1) ^ rc1) \vee (qc1 ^ rc1)) = T
```

to prove the subgoal: rc1 = T == true

```
-LP script-------------------------------------------------------------------------------
Lemma cancel_lemma.1: Subgoal for proof of =>
New constants: pc1,  rc1,  qc1
Hypothesis:
  cancel_lemmaImpliesHyp.1:
     ((nnot(pc1) ^ rc1) \vee (qc1 ^ rc1)) = T == true
Subgoal:
  cancel_lemma.1.1: rc1 = T == true
-----------------------------------------------------------------------------------------
```

4.2 An Induction principle for leads_to

The following induction rule has a crucial importance in the proof system because it involves the most interesting predicate, namely *leads_to*. Remember that this predicate helps to specify the progress properties of a program. In most of the proofs of concurrent programs, in order to prove a progress property, we have to exhibit something which "decreases", so we need a rule to specify that fact. We give the rule as it was in [CM88a]:

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

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 the 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 among list of arithmetic expression. We define this ordering as follow:

lexico(nil,nil)==T
lexico(cons(e1,list1),cons(e2,list2))==(e1<e2) $\vee$ ((e1=e2)$\wedge$ lexico(list1,list2))

where list1 and list2 are two list of expression, e1 and e2 two expressions. We define $<: exp, exp \rightarrow bexp$ as an extention of $<: nat, nat \rightarrow bool$

The formalization of the induction principle is :

```
Set name induc_princ
assert
leads_to(p ∧ equal_exp_list(exp_list1,exp_list2),
         (p ∧ lexico(exp_list1,exp_list2)) ∨ 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 :

$$\textbf{IND\_coroll}: \quad \frac{\mathsf{leads\_to}(p \wedge \mathsf{eq}(l1, l2), \mathsf{lexico}(l1, l2), \mathsf{pgm})}{\mathsf{leads\_to}(\mathsf{true}, \neg p, \mathsf{pgm})}$$

The reader can find examples based on this work in [Che95], where we describe a fully computer checked proof of a UNITY program. The example is a protocol for communications over faulty chanels, provided by Chandy and Misra in [CM88a].

5 Discussion

In this paper, we have described the mechanization of UNITY logic and of its proof system, and we have shown that the translation into the first-order logic of LP is sufficiently natural to allow concentrating efforts on formal proofs.

In fact this simplicity is obtained by a *sophistication* of the encoding. The axiomatization of the standard data types we used, such *naturals* and *sequences*, are part of the Larch Shared Language (LSL) [GHG+93]. We have modified these definitions in order to axiomatize customized data types and to produce formal definitions for basic concepts related to the case studies. On the other hand, additional work was required to simplify the translation and to bring some expressions into a form that the prover can handle.

We have seen that the style of specification in LP can be used to specify UNITY, since algebraic specifications are easy to write and to use. This is important when dealing with concurrent programs whose proofs are complex in essence. It is important to note that the complexity of the specifications comes from the embedding of sorts, which made it difficult to read. Therefore we concentrated our efforts first to make it as readable as possible by a correspondence between function names and their semantics. This problem could be overcome with future developments of the prover.

In this paper, thanks to the wp_calculus, we were able to reason on the syntactic structure of the program itself instead of reasoning on the computations. Indeed never in the course of the proof we made any mention to a run of the program, unlike Goldshlag in [Gol90] in his NQTHM proof. Our approach was closer to this of Brown and Mery in [BM93].

Our future efforts will concentrate on the mechanization of the use of invariants in the proofs. In particular we will investigate how we can use a proved invariant in a proof. Indeed presently, we introduce these invariants as axioms in the proofs. On the other hand, we want to reduce the script of the proofs with less directives to the prover. One way to reduce the

script of a proof is to use more deduction rules, because it is the only way to infer facts without interacting with the user. But, as explained before, that may lead to infinite loops, and even if not, the use of many deduction rules slows down the proof process.
Another issue of our work will be the design of a translator, which translates automatically a UNITY program in a script LP, and proof obligations as input to LP.

In [B.A91], the author shows that combining the substitution axiom [CM88a] with the three temporal predicates leads to an unsound proof system. In our system, we do not code the substitution axiom, and we are not concerned by the *common misunderstanding* about the definition of the operators *unless* and *ensures* [B.A91].

6 Conclusion

Our experiences with the theorem prover LP, to prove a circuit of division [CL92] and attempts done by others to use it in the verification of circuit design or for the verification of a compiler [Sco92], ..., made us confident to try it in the proof of concurrent programs. We chose UNITY as a proof environment for concurrent programs, because of it success to provide a common foundation for the design of programs for a variety of parallel and distributed architectures.
The course of our investigation leads us to make the mechanization of LARCH proof of UNITY properties as practical and natural as possible in order to be able to formally verify without excessive effort correctness of programs expressed in this formalism. Despite many researchers urge for the most powerful prover, we were happy with an assistant provided it helped us to find easily bugs in wrong proofs and it allowed us to express properties we wanted to prove about individual programs.
This experiment has been conduct with the release 2.4 of LP and our proof system contains about 500 rewrite rules, including the definitions of *naturals*, *sequences*, *boolean expressions* ..., and the theorems about the temporal predicates.

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. Chandy and Misra seem to have captured what is important for the design of parallel programs and provide a program notation, specification language and proof theory in order to specify and to prove in a clearly and concise way. So it is important to do that "automatically" and to get formal verified proofs.

References

[And92] F. Andersen. *A theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

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

[BM93] N. Brown and D. Mery. A proof environment for concurrent programs. In *In Proceedings FME93 Symposium*, volume 670 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Che95] B. Chetali. A formal proof of a protocol for communications over faulty channels using the larch prover. Research Report 2476, Inria-Lorraine, 1995.

[CL92] B. Chetali and P. Lescanne. An exercise in LP, the proof of a non restoring division circuit. In U. Martin and J. Wing, editors, *Proc. of First International Workshop on Larch*. Springer-Verlag, 1992.

[CM88a] K. M. Chandy and J. Misra. *Parallel Program Design: A Fundation*. Addison-Wesley, 1988. ISBN 0-201-05866-9.

[CM88b] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*
A Foundation. Addison-Wesley, 1988.

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

[GG91] S. V. Garland and J. V. Guttag. A guide to lp, the larch prouver. Technical Report 82, Digital Ssystems 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.

[Gol90] D. M. Goldschlag. Mechanically verifying concurrent programs with the boyer-moore prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1022, September 1990.

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

[Sco92] E. A. Scott. Using LP to investigate compiler correctness. Technical report, IMEC-2.A.2-01 of CHARME BRA 3216, 1992.

[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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399