INSTITUT DE RECHERCHE EN INFORMATIQUE DE TOULOUSE

IRIT-97-44-R

Quantifier elimination technics for program validation

J.-P. Bodeveix, M. Filali

CNRS (UMR 5505) - INP - UPS

Quantifier elimination technics for program validation

J.-P. Bodeveix, M. Filali
{bodeveix,filali}@irit.fr
IRIT-Université Paul Sabatier
118 Route de Narbonne
F-31062 Toulouse cédex
France

October 22, 1997

Résumé

Cet article présente des techniques d'automatisation de la vérification de programmes dont l'espace d'état est infini ou paramétré. Pour cela, nous étudions la transformation de formules du second ordre exprimant la conservation d'invariants en des formules équivalentes mais exprimées dans une logique plus faible mais décidable. Deux techniques sont abordées: l'élimination de quantificateur et la réduction à un domaine fini. Nous illustrons ces techniques par la validation de protocoles de cohérence mémoire.

Abstract

We study the automation of the verification of programs of which state space is infinite or parameterized. This paper presents methods allowing the transformation of some second-order formulas into equivalent formulas expressed in a weaker but decidable logic. Two technics are considered: quantifier elimination and reduction to finite domain. We illustrate these technics on the validation of memory coherency protocols.

Contents

1 Introduction

In the last few years, many program validation tools have been developed [Hol91, MAB+94]. Their aim is to check temporal properties of concurrent programs. In general, these tools can automate the verification process when the program state space is finite. However, if the program space is either infinite or parameterized, the problem becomes undecidable in general.

In this paper, we study how such an automation can be extended to either infinite state programs, or families of programs. To reduce the complexity of the considered problem, we only consider safety properties. The elimination of the temporal dimension allows us to decide some problems about infinite state spaces. The decision for the obtained formulas relies on the reduction of their order. Mainly, we study how to convert second order formulas into equivalent ones expressed in a weaker decidable logic. Two technics are considered: quantifier elimination and reduction to a finite domain.

In section 2, we state more precisely the problem in the context of Unity programs [CM88]. In section 3, we review some existing results concerning quantifier elimination. In section 4, we propose technics using unskolemization and in section 5 technics using domain reduction. In the last section, we present two case studies to illustrate the presented technics.

2 Program semantics

There exists many ways for representing Unity program semantics. Such a representation depends on the level of reasoning: either we are at the language semantic level, or we are at the program semantic level. These levels differ mainly by the knowledge of the program variables and their types. At the language semantic level, although several approaches have been proposed, it remains tricky to ensure a strict correspondence between variables and their types [And92, BF95]. At the program semantics level, things are easier: actually the à priori knowledge of the program variables and their types allows a definition of a dedicated semantic function. Since in this paper we are concerned by properties of given programs, we consider the program semantic level.

2.1 Program state

The program state is represented by a tuple. Each element of this tuple represents the value of a program variable. The representation of the value of a scalar variable is typed as the variable; the representation of the value of an array variable is a function with compatible domain and range. If needed, we can introduce new state variables for array bounds.

Example: let us consider the following declaration extracted form the Illinois protocol.

```
type
  State = {Inv, Excl, Shared, Dirty};
var
  Ch  : array Proc of Word;    --  private copy (per processor) of memory data
  Ctr : array Proc of State;   --  state (per processor) of the cache
  M   : Word;                  --  memory
```

The representation of the data structures needs the introduction of the user defined types. In the HOL [GM94] syntax, we write:

```
new_type {Arity = 0, Name = "Proc"};
new_type {Arity = 0, Name = "Word"};

val State = define_type {
  name = "State",
  type_spec = `State = Inv | Excl | Shared | Dirty`,
  fixities = [Prefix,Prefix ,Prefix ,Prefix ]};
```

Then the tuple representing the program state is typed as follows:

```
(Proc->Word) # (Proc->State) # Word
```

Thus, a variable is represented by an access function to the corresponding element of the tuple. For instance, the variable `C` is defined by `C(st) = FST(st)`.

2.2 Statements logic

A Unity program is defined as an alternative between a set of conditional multi-assignments. Since in this paper we only consider safety properties it is sufficient to reason locally at the basic statement level. For instance, in order to verify a stable property we need to prove that it is preserved by each statement. Furthermore, although conditional multi-assignments are deterministic statements, we encode them as binary relations over couples of states. This approach allows to deal as well with partially defined statements (array assignments) as well with non-deterministic statements (as TLA [Lam94] actions).

For each elementary statement, we separate the guard from the action, thus producing, in HOL, the following definition schema. In such a definition, the relation is parameterized by the variables p occurring in quantified alternatives[1].

```
val u_action = new_definition("u_action",
  ``u_action p s s' = ((guard p s) => (action p s s') | (s' = s))``
);
```

In the following, for the sake of readability, we omit the parameter p. The `action` predicate expresses the semantics of a parallel statement. It is defined as the conjunction of relations action_v expressing the update of each state variable $v \in \mathcal{V}$:

$$\text{action}(s, s') = \bigwedge_{v \in \mathcal{V}} \text{action}_v(s, s')$$

The general schema of a parallel statement is $S_0 \parallel \ldots \parallel S_n$ where S_i are simple or parallel conditional assignments. We can reorder the statements into $A_{v_0} \parallel \ldots \parallel A_{v_m}$ where A_{v_i} contains all the assignments to the state variable v_i. Then, we distinguish scalar and array variables:

- If v is a scalar variable, we have: $A_v \equiv v := e$ **if** c. The semantics of this conditional assignment is expressed by the following relation between the global states s and s':

$$\text{action}_v(s, s') = (v(s') = \textbf{if } c(s) \textbf{ then } e(s) \textbf{ else } v(s))$$

[1] $c \to p \mid q$ denotes the conditional expression **if** c **then** p **else** q.

- If v is an array variable, we have:

$$
\begin{array}{rll}
A_v \equiv & & \langle \| \; q :: v[n_1(q)] := e_1(q) \textbf{ if } c_1(q) \rangle \\
& \| & \ldots \\
& \| & \langle \| \; q :: v[n_k(q)] := e_k(q) \textbf{ if } c_k(q) \rangle
\end{array}
$$

  Note that if the sequence of quantified variables q is empty, we get an assignment to a single element of the array v.

  The semantics of the multiple conditional assignment is also expressed by a relation between the global states s and s'[2]:

$$
\begin{array}{rl}
\text{action}_v(s, s') = & \\
\forall x.\ v(s', x) = & \textbf{if } (\exists q.\ x = n_1(q, s) \wedge c_1(q, s)) \textbf{ then } e_1(\varepsilon q.\ x = n_1(q, s) \wedge c_1(q, s), s) \\
& \ldots \\
& \textbf{elsif } (\exists q.\ x = n_k(q, s) \wedge c_k(q, s)) \textbf{ then } e_k(\varepsilon q.\ x = n_k(q, s) \wedge c_k(q, s), s) \\
& \textbf{else } v(s, x)
\end{array}
$$

If a state variable v is not assigned, the relation action_v is defined as:

- If v is a scalar variable: $\text{action}_v(s, s') = (v(s') = v(s))$
- If v is an array variable: $\text{action}_v(s, s') = (\forall x.\ v(s', x) = v(s, x))$

In fact, for each state variable v, we have the following schema:

$$\text{action}_v(s, s') = (v(s') = \varphi_v(s))$$

Thus, the action predicate looks like:

$$\text{action}(s, s') = ((v_0(s'), \ldots, v_n(s')) = (\varphi_{v_0}(s), \ldots, \varphi_{v_n}(s))$$

2.3 Validation of invariants

In this section, we describe the generic formulas which we are interested to prove and the technics we propose. Since we only study the automatic verification of Unity invariants, we have to verify that a predicate is true initially and preserved by each basic statement. In fact, we should also check the existence of an initial state. In the following, we illustrate the proposed technics on the invariant preservation.

2.3.1 The generic formula

According to the previous section, the Unity basic statements have a functional interpretation, and their semantics and thus the validation of an invariant leads to the proof of formulas of the following pattern:

$$\forall v_1 \ldots v_n.\ P(v_1, \ldots, v_n) \Rightarrow Q(\varphi_1(v_1, \ldots, v_n), \ldots, \varphi_n(v_1, \ldots, v_n)) \tag{1}$$

[2] ε denotes the choice operator

where the state space has been represented by the tuple $(v_1, \ldots, v_n)$ and a statement by a tuple $(\varphi_1, \ldots, \varphi_n)$. As some v_i are functions, the formula (1) is in general not decidable. In order to prove automatically such a formula, we look for an equivalent one which can be expressed in a decidable fragment of the logic. Basically, we consider the S1S and the WS1S fragments [Tho90], and propose technics for replacing a quantification over natural-valued functions by quantifications over unary predicates, naturals or booleans. Then, provided the initial predicates P and Q are S1S, the proposed equivalent formula belongs also to the S1S fragment and consequently is decidable.

With respect to formula (1), we isolate each quantification over such variables and consider the basic formula $\forall v.\ P(v)$ where v is a functional variable.

2.4 Working example: the Illinois protocol

The Illinois protocol [AB86] is a cache coherency algorithm for shared memory multi-processors. Each processor owns a private cache where it stores copies of global memory data. The algorithm enforces the coherency between multiple copies of the same data.

Specification The data structure of this algorithm can be described as follows:

```
type
  State = {Inv, Excl, Shared, Dirty};
var
  Ch  : array Proc  of Word;    --  private copy (per processor) of memory data
  Ctr : array Proc  of State;   --  state (per processor) of the cache
  M   : Word;                   --  memory
```

The safety of the algorithm can be specified by the following invariant expressed as the conjunction of three basic properties:

```
invariant
  ∀ p q. (Ctr[p] ≠ Inv ∧ Ctr[q] ≠ Inv) ⇒ (C[p] = C[q]) { I0 }
  ∀ p. (Ctr[p] ∈ {Excl, Shared}) ⇒ (C[p] = M) { I1 }
  ∀ p q. (Ctr[p] ∈ {Excl, Dirty} ∧ Ctr[q] ≠ Inv) ⇒ (p = q) { I2 }
```

where

- I0 asserts that two non invalid caches contain the same data.
- I1 asserts that a shared or exclusive cache is coherent with the memory.
- I2 asserts that a cache which is exclusive or dirty is also the only non invalid cache.

Validation example Consider the following read statement:

```
⟨ ‖ p, q :: Ctr[p] = Inv ∧ Ctr[q] ∈ {Shared,Excl} →
      ⟨ || r :: Ctr[r] := Shared if (r = p) ∨ (Ctr[r] ≠ Inv) ⟩
   || C[p] := C[q]
⟩
```

The validation of the invariant on such a statement requires the validation of the following formula:

```
∀ C Ctr M p q. Inv(C,Ctr,M) ∧ g_IllinoisRead(p,q,C,Ctr,M) ⇒
        Inv(λ u. if p = u  then C(q) else C(u),
            λ u. if (u = p) ∨ (Ctr(u) ≠ Inv) then Shared else Ctr(u),
            M)
```

where `Inv` and `g_IllinoisRead` are defined as:

```
Inv(C,Ctr,M) = I0(C,Ctr,M) ∧ I1(C,Ctr,M) ∧ I2(C,Ctr,M)
g_IllinoisRead(p,q,C,Ctr,M) = (Ctr(p) = Inv) ∧ Ctr(q) ∈ {Excl, Shared}
```

The main purpose of the proposed technics will be to eliminate the quantification over `C`. After this step, the obtained formula contains quantifications over unary predicates (introduced by the elimination process), finite valued functions (`Ctr`) and variables ranging over the finite domain `Proc`. Then, such a formula belongs to the S1S logic fragment: it is decidable.

2.5 About the use of model checkers

In the last few years, several model checkers have shown that they can handle very large state spaces [BCMD90]. Moreover, although it is acknowledged that their use is much more easy than the use of theorem provers, the problems we are dealing with cannot be resolved through the exclusive use of model checkers. Actually, we consider problems where some data structures are not fixed, à priori. For instance, when we consider a cache coherency problem we do not want to fix neither the number of processors, nor the bus width; another example of such a structure is in communication protocols where we do not want to fix a length for the communication queues. In fact, we want to carry a proof that should be valid not only for one problem, but for a whole family of problems; so in the statement of our initial problem we do not have a finite state space and we cannot use model checkers.

However, as we will see in section (5) some of the transformations we propose are based on domain reduction. The obtained domains being finite after such reductions, we can apply model checking technics to decide the transformed formulas.

3 Overview of quantifier elimination technics

3.1 Unskolemization

Since most of the presented methods are based on unskolemization, we first recall the basic unskolemization theorem:

$$\exists f.\forall x.P(x,f(x)) = \forall x.\exists y.P(x,y)$$

It is interesting to remark that this is a basic quantification elimination over functions. Actually, a second order quantification (over f) has been replaced by a first order quantification (over y).

3.2 Overview of Ackermann

The Ackermann's technics are among the first that were developed to solve the elimination problem [Ack68]. In terms of λ calculus, these results can be stated as the the two following theorems:

Theorem 1 *Let B be a predicate which contains only positive occurrences of φ, what is expressed as follows:*

$$\forall \varphi_1 \varphi_2.\ (\forall\ x.\ \varphi_1(x) \Rightarrow \varphi_2(x)) \Rightarrow (B(\varphi_1) \Rightarrow B(\varphi_2))$$

Then, we have the following elimination results, where the second one is derived from the first one by substituting φ by $\neg\varphi$ in the body of the second order formula:

$\vdash$ **Ackermann-1** $(\exists \varphi.\ \forall x.(\varphi(x) \vee A(x,z)) \wedge B(\lambda x.\neg\varphi(x))) = B(\lambda x.A(x,z))$

$\vdash$ **Ackermann-2** $(\exists \varphi.\ \forall x.(\neg\varphi(x) \vee A(x,z)) \wedge B(\varphi)) = B(\lambda x.A(x,z))$

An interesting application of these technics was developed for the DLS algorithm (see section 3.4).

Example Let us consider the following formula from [DLS95]:

$$\exists P\ \forall y\ (\neg R(x,y) \vee P(y)) \wedge (\exists y\ (R(x,y) \wedge \exists z\ (R(y,z) \wedge \neg P(z))))$$

Applying Ackermann's lemma yields to the following equivalent first-order formula:

$$\exists y\ (R(x,y) \wedge \exists z\ (R(y,z) \wedge \neg R(x,z)))$$

3.3 Overview of Scan

The Scan algorithm [GO92] is based on first order resolution to eliminate second order quantified predicate variables. Its basic idea is first to put the formula into clausal form, using skolemization if needed, second to iterate the computation of all the so-called C-resolvents and C-factors of the clausal form of the formula, according to the following rules:

$$\textbf{(C} - \textbf{resolvent)}$$

$$\left.\begin{array}{r} \phi(s_1,\ldots,s_n) \vee C \\ \neg\phi(t_1,\ldots,t_n) \vee D \end{array}\right\} \quad \Rightarrow \quad C \vee D \vee \bigvee_{i=1,n} s_i \neq t_i$$

$$\textbf{(C} - \textbf{factor)}$$

$$\phi(s_1,\ldots,s_n) \vee \phi(t_1,\ldots,t_n) \vee C \quad \Rightarrow \quad \phi(s_1,\ldots,s_n) \vee C \vee \bigvee_{i=1,n} s_i \neq t_i$$

If this step terminates, we put back the quantified prefix in front of the conjunction of clauses not containing ϕ and try to unskolemize. If it succeeds, we get a first-order formula equivalent to the initial one.

Example Let us start from a formula containing the second-order variable P and first-order variables, for instance:

$$\exists P\ \forall x,y\ \exists z (\neg P(a) \vee Q(x)) \wedge (P(y) \vee Q(a)) \wedge P(z)$$

After skolemization of z by the function f, we get the quantified prefix $\exists P, f\ \forall x, y$ and the following clausal form

$$\left\{\begin{array}{l} \neg P(a) \vee Q(x) \\ P(y) \vee Q(a) \\ P(f(x,y)) \end{array}\right.$$

The computation of all possible resolvents leads to the two following clauses (after some simplifications):

$$\begin{cases} Q(a) \\ a \neq f(x,y) \vee Q(x) \end{cases}$$

Then after the elimination of the clauses containing P, restoration of the quantifiers, and unskolemization, we obtain the result:

$$\forall x \exists z\ Q(a) \wedge (a \neq z \vee Q(x))$$

Remark The application of Scan in a restricted context can be easily expressed in higher order logic. If each clause only contains one occurrence of P, and if P appears positively in only one clause, in fact Scan applies the following theorem:

$$(\exists P.(\forall x.P(t(x)) \vee C(x)) \wedge (\forall x. \textstyle\bigwedge_i \neg P(f_i(x)) \vee D_i(x)))$$
$$=$$
$$(\forall x\ y. \textstyle\bigwedge_i (t(x) \neq f_i(y)) \vee C(x) \vee D_i(y))$$

3.4 Overview of DLS

As Scan, the DLS algorithm [DLS95] reduces second-order formulas to logically equivalent first-order formulas. However, DLS attempts to apply either Ackermann's results or the following derived theorem on a conjunctive normal form of the formula:

Theorem 2 *Let A be a formula of the form pref($A_1 \wedge \ldots \wedge A_q$), where pref is a prefix of first-order quantifiers and $A_1, \ldots, A_q$ are disjunctions of literals. In addition, let ϕ be a predicate variable occurring in A and Conj(A) those conjuncts in A where ϕ occurs. Assume that for any conjunct in Conj(A), ϕ occurs either positively or both positively and negatively (or analogously, negatively, or both negatively and positively). Then*

$$\exists \phi A = pref(A_{i_1} \wedge \ldots \wedge A_{i_r})$$

where $i_1, \ldots, i_r \in \{1, \ldots, q\}$ and $A_{i_1}, \ldots, A_{i_r}$ are all the conjuncts that do not contain occurrences of ϕ (the empty conjunction is regarded to be equivalent to $\top$).

The main steps of the DLS algorithm are the following:

- Preprocessing: the aim of this step is to isolate positive and negative occurrences of ϕ so that and equivalent formula as the following is obtained:

  $$\exists \overline{x}(\exists \phi(A_1(\phi) \wedge B_1(\phi))) \vee \ldots \vee (\exists \phi(A_n(\phi) \wedge B_n(\phi)))$$

  where A_i (resp. B_i) contains only positive (resp. negative) occurrences of ϕ

- Preparation for Ackermann: some skolemizations may be needed for applying it.

- Application of one of Ackermann's results and unskolemization if possible.

- Simplification.

We remark that DLS always terminates and can simplify some formulas on which the Scan algorithm loops. Actually, Scan cannot manage a formula where a disjunct contains both a positive and a negative occurrence of the predicate to be eliminated.

Example

$$
\begin{array}{cl}
 & \forall\Phi(\exists x.\ \Phi(x) \Rightarrow \forall x.\ \Phi(x)) \\
= & \{\textit{Existential form}\} \\
 & \neg\exists\Phi(\exists x.\ \Phi(x) \wedge \exists x.\ \neg\Phi(x)) \\
= & \{\textit{Factorization of existential quantifiers}\} \\
 & \neg\exists\Phi.\ \exists x\ y.\ (\Phi(x) \wedge \neg\Phi(y)) \\
= & \{\textit{Extraction of existential quantifiers}\} \\
 & \neg\exists x\ y.\ \exists\Phi.\ (\Phi(x) \wedge \neg\Phi(y)) \\
= & \{\textit{Generalization of } \Phi \textit{ arguments}\} \\
 & \neg\exists x\ y.\ \exists\Phi.\ (\forall u.\ u \neq x \vee \Phi(u) \wedge \forall u.\ u \neq y \vee \neg\Phi(u)) \\
= & \{\textit{Factorization of universal variables}\} \\
 & \neg\exists x\ y.\ \exists\Phi.\ \forall u.\ (u \neq x \vee \Phi(u) \wedge u \neq y \vee \neg\Phi(u)) \\
= & \{\textit{Ackermann-1 with } A(u,x) = u \neq x \textit{ et } B(\Phi) = u \neq y \vee \Phi(u)\} \\
 & \neg\exists x\ y.\ \exists\Phi.\ \forall u.\ (u \neq y \vee u \neq x) \\
= & \{\textit{Simplifications}\} \\
 & \forall x\ y.\ y = x
\end{array}
$$

4 Unskolemization-based technics

4.1 Functions applied on universal variables

This result transforms the quantification over a function f only applied to universal variables into quantifications over variables representing their images as expressed by the following equivalence:

$$
\begin{array}{c}
\forall f\ x_1 \ldots x_n.\ P(x_1, f(x_1), \ldots, x_n, f(x_n)) \\
= \\
\forall x_1 \ldots x_n, y_1 \ldots y_n.\ (\forall x.\exists y.\ \bigwedge_{i=1,n}(x = x_i \Rightarrow y = y_i)) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n)
\end{array}
\tag{2}
$$

The proof of the preceding formula relies mainly of the unskolemization theorem:

$$
\begin{array}{cl}
 & \forall f\ x_1 \ldots x_n.\ P(x_1, f(x_1), \ldots, x_n, f(x_n)) \\
= & \{\textit{Generalization of the arguments of } P\} \\
 & \forall f\ x_1 \ldots x_n, y_1 \ldots y_n.(\bigwedge_{i=1,n}(y_i = f(x_i))) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n) \\
= & \{\textit{Restriction of the scope of } f\ \} \\
 & \forall\ x_1 \ldots x_n, y_1 \ldots y_n.(\exists f.\ \bigwedge_{i=1,n}(y_i = f(x_i))) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n) \\
= & \{\textit{Generalisation of the arguments of } f\} \\
 & \forall\ x_i, y_i.\ (\exists f.\ \bigwedge_{i=1,n}(\forall x.(x = x_i) \Rightarrow y_i = f(x))) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n) \\
= & \{\textit{Extraction of universal quantifiers}\} \\
 & \forall\ x_i, y_i.\ (\exists f.\ \forall x.\ \bigwedge_{i=1,n}((x = x_i) \Rightarrow f(x) = y_i)) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n) \\
= & \{\ \textit{Unskolemization}\} \\
 & \forall x_i, y_i.\ (\forall x.\exists y.\ \bigwedge_{i=1,n}(x = x_i \Rightarrow y = y_i)) \Rightarrow P(x_1, y_1, \ldots, x_n, y_n)
\end{array}
$$

Remark The preceding result can be applied successively to eliminate m functional variables. This elimination introduces $m \times n$ first-order variables y_i.

4.2 Functions compared with universal variables

This result transforms a formula quantified over a function f into a formula quantified over predicates with the same arity as f. This transformation can be applied when the result of f is compared with universal variables. We state this result as follows:

$$\begin{array}{c}\forall f\ y_1 \dots y_n.\ P(y_1, \lambda x.f(x) = y_1, \dots, y_n, \lambda x.f(x) = y_n) \\ = \\ \forall y_1 \dots y_n\ p_1 \dots p_n.\ (\forall x.\exists y. \bigwedge_{i=1,n}(p_i(x) = (y = y_i))) \Rightarrow P(y_1, p_1, \dots, y_n, p_n)\end{array} \tag{3}$$

Hence, if the initial function f is unary and provided that P is S1S, the resulting formula can be embedded in the S1S fragment and therefore be decided. Again, the proof of this result mainly relies on the unskolemization theorem:

$$\begin{array}{ll} & \forall f\ y_1 \dots y_n.\ P(y_1, \lambda x.f(x) = y_1, \dots, y_n, \lambda x.f(x) = y_n) \\ = & \{\ \textit{Generalization of the arguments of } P\ \} \\ & \forall f\ y_1 \dots y_n\ p_1 \dots p_n.(\bigwedge_{i=1,n}(p_i = \lambda x.f(x) = y_i)) \Rightarrow P(y_1, p_1, \dots, y_n, p_n) \\ = & \{\ \textit{Restriction of the scope of } f\ \} \\ & \forall\ y_i\ p_i.(\exists f.(\bigwedge_{i=1,n} \forall x.p_i(x) = (f(x) = y_i))) \Rightarrow P(y_1, p_1, \dots, y_n, p_n) \\ = & \{\ \textit{Extraction of universal quantifiers}\ \} \\ & \forall\ y_i\ p_i.(\exists f.\forall x. \bigwedge_{i=1,n}(p_i(x) = (f(x) = y_i))) \Rightarrow P(y_1, p_1, \dots, y_n, p_n) \\ = & \{\ \textit{Unskolemization}\ \} \\ & \forall y_1 \dots y_n\ p_1 \dots p_n.\ (\forall x.\exists y. \bigwedge_{i=1,n} p_i(x) = (y = y_i)) \Rightarrow P(y_1, p_1, \dots, y_n, p_n)\end{array}$$

Remark The preceding result can be applied successively to eliminate m functional variables. This elimination introduce $m \times n$ predicate variables p_i.

4.3 Another function elimination technic

In this section, we propose an algorithm to transform a second-order formula into a first-order one, which is based on unskolemization. Let us consider the following second order formula:

$$\exists f.P(f) \tag{4}$$

where $P(f)$ is a classical first order formula. The proposed algorithm applies the following steps: normalization, argument generalization and unskolemization.

4.3.1 Normalization step

In this step, we try to transform (4), using the same basic rewrites as DLS, into a disjunction of formulas such as:

$$\exists \overline{u}.\ \exists\ f.\ P_1 \wedge \dots \wedge P_n \tag{5}$$

where the P_i have one of the following patterns:

P1 $\forall \overline{x}.\ P(f(g(\overline{x})))$ where g does not use any variable bound in P[3].

[3] We should have: $(\forall \overline{x}.\ P(f(g(\overline{x})))) = (\lambda g.\ \forall \overline{x}.\ P(f(g(\overline{x})))g$

P2 $\forall xy.\ R(x) \vee R(y) \vee (f(x) = f(y))$

P3 φ where the arguments of each occurrence of f in φ only contain free variables with respect to φ.

Example: let us consider the following formula (obtained in the elimination process of the case study of §6) where C is the function to be eliminated:

```
¬(∀C Ctr M.
       ((∀u1 u2. P Ctr u1 ∧ P Ctr u2 ⇒ (C u1 = C u2)) ∧
        (∀u1. Q Ctr u1 ⇒ (C u1 = M))) ∧
       R Ctr ⇒
       (∀u1 u2. P (f_ctr Ctr) u1 ∧ P (f_ctr Ctr) u2 ⇒
         (f_c Ctr u1 (C (f_c1 Ctr u1)) M = f_c Ctr u2 (C (f_c1 Ctr u2)) M)))
```

The normalization step produces the following equivalent formula:

```
(∃ Ctr. R Ctr ∧                                                          (4.3.1)
    (∃ M u1. P (f_ctr Ctr) u1 ∧
      (∃ u2. P (f_ctr Ctr) u2 ∧
        (∃ C. (∀ u3 u4. ¬(P Ctr u3) ∨ ¬(P Ctr u4) ∨ (C u3 = C u4)) ∧
              (∀ u5. ¬(Q Ctr u5) ∨ (C u5 = M)) ∧
              (f_c Ctr u1 (C (f_c1 Ctr u1)) M ≠ f_c Ctr u2 (C (f_c1 Ctr u2)) M)))))
```

where the sub-formula existentially quantified by `C` is a conjunction of three formulas satisfying one of the patterns **P1** to **P3**.

Algorithm For this purpose, as in the preprocessing step of DLS [DLS95], we apply the following transformations:

- eliminate the $\Rightarrow$ and $\Leftrightarrow$ connectives
- move the negation connective to the right until all its occurrences immediately precede atomic formulas
- move existential quantifiers to the left
- repeat the two following steps while some change occurs
  - move universal quantifiers to the right
  - distribute disjunctions over conjunctions.

4.3.2 Argument generalization step

The goal of this step is to universally quantify all the arguments of the function f to be eliminated. Each pattern produced by the normalization step is transformed into: $\exists \overline{v}.\forall x.\ P(\overline{v}, x, f(x))$. Then, existentially quantified variables are lifted up and put ahead (some renaming may be necessary). The resulting formula then looks like:

$$\begin{array}{lll} \exists \overline{w} \exists f. & & \forall x.\ P_1(\overline{w}, x, f(x)) \\ & \wedge & \ldots \\ & \wedge & \forall x.\ P_n(\overline{w}, x, f(x)) \end{array}$$

We now describe the transformation rules for each pattern:

P1 $\forall \overline{x}.\ P(f(g(\overline{x}))) = \forall \overline{x}\ y.\ (y = g(\overline{x})) \Rightarrow P(f(y))$

P2 $\forall xy.\ R(x) \vee R(y) \vee (f(x) = f(y)) = \exists e.\ \forall x.\ R(x) \vee f(x) = e$

P3 φ: For this purpose, the following rewriting theorem is applied recursively on all occurrences of f:

$$\varphi(f(t)) = (\exists\ y.\ (\forall x.(x = t) \Rightarrow (f(x) = y)) \wedge \varphi(y)) \tag{6}$$

After these rewrites, existential quantifiers are extracted and put behind the existential quantification of f.

Example: let us consider the following subformula of the formula obtained after the normalization of the memory example (4.3.1). The following rewrites illustrate the application of **P3** rule, C being the function to be eliminated:

```
 f_c(Ctr,u1,C(f_c1(Ctr,u1)),M) ≠ f_c(Ctr,u2,C(f_c1(Ctr,u2)),M)
= { (6) }
 ∃ y1. ∀ x1. (x1 = f_c1(Ctr,u1)) ⇒ (C(x1) = y1) ∧
       f_c(Ctr,u1,y1,M) ≠ f_c(Ctr,u2,C(f_c1(Ctr,u2)),M)
= { (6) }
 ∃ y1. ∀ x1. (x1 = f_c1(Ctr,u1)) ⇒ (C(x1) = y1) ∧
       ∃ y2. ∀ x2. (x2 = f_c1(Ctr,u2)) ⇒ (C(x2) = y2) ∧
             f_c(Ctr,u1,y1,M) ≠ f_c(Ctr,u2,y2,M)
= { Extraction of existential quantifiers }
 ∃ y1 y2. ∀ x1. (x1 = f_c1(Ctr,u1)) ⇒ (C(x1) = y1) ∧
          ∀ x2. (x2 = f_c1(Ctr,u2)) ⇒ (C(x2) = y2) ∧
          f_c(Ctr,u1,y1,M) ≠ f_c(Ctr,u2,y2,M)
```

4.3.3 Unskolemization step

After the factorization of the universal quantifiers produced by the previous step, unskolemization applies and we have:

$$\exists \overline{w}\ \exists f.\ \left(\begin{array}{ll} & \forall x.\ P_1(\overline{w}, x, f(x)) \\ \wedge & \ldots \\ \wedge & \forall x.\ P_n(\overline{w}, x, f(x)) \end{array}\right) = \exists \overline{w}.\ \forall x.\ \exists y.\ \left(\begin{array}{ll} & P_1(\overline{w}, x, y) \\ \wedge & \ldots \\ \wedge & P_n(\overline{w}, x, y) \end{array}\right)$$

4.3.4 Comments

Our study has outlined three basic elimination rules (**P1 P2 P3**). The rule **P2** may seem tricky; however this pattern actually appears in concrete examples. One could explain such by the fact that in many problems the basic property can be specified as a boundedness one. For instance, **P2** could be reformulated as "at most one element of image of f verifies $\neg R$", which can be written Atmost$(1, f, \neg R)$.

In fact, we can generalize the **P2** case to `Atmost(n,f,D)` and `Atleast(n,f,D)`. Actually, these two dual predicates admit an existential as well as a universal expression. Such a versatility makes easy the transformation of either a positive, or negative occurrence into an existential formula which can be unskolemized. More precisely:

$$
\begin{array}{l}
\mathrm{Atmost}(n, f, D) \\
\ = \exists y_1, \ldots, y_n.\ \underline{\forall x}.\ D(x) \Rightarrow \bigvee_i \underline{f(x)} = y_i \\
\ = \forall x_1, \ldots, x_{n+1}.\ \bigwedge_i D(x_i) \Rightarrow \bigvee_{i<j} f(x_i) = f(x_j) \\
\ = \forall x_1, \ldots, x_{n+1} (y_{ij})_{i<j}.\underline{\exists u}.\ \bigwedge_i D(x_i) \Rightarrow \bigvee_{i<j} (\underline{f(u)} = y_{ij}) \rightarrow u = x_j \mid u = x_i \\
\mathrm{Atleast}(n+1, f, D) \\
\ = \neg \mathrm{Atmost}(n, f, D) \\
\ = \forall y_1, \ldots, y_n.\ \exists x.\ D(x) \wedge \bigwedge_i f(x) \neq y_i \\
\ = \exists x_1, \ldots, x_{n+1}.\ \bigwedge_i D(x_i) \wedge \bigwedge_{i<j} f(x_i) \neq f(x_j) \\
\ = \exists x_1, \ldots, x_{n+1} (y_{ij})_{i<j}.\underline{\forall u}.\ \bigwedge_i D(x_i) \wedge \bigwedge_{i<j} (\underline{f(u)} = y_{ij}) \rightarrow u \neq x_j \mid u \neq x_i
\end{array}
$$

4.3.5 Comparison with Scan and DLS

A major difference between the elimination result we have proposed and those of Scan and DLS is that they deal with predicates while we deal with unary functions. Moreover, technically, Scan is based on a calculus of resolvents and DLS is based on the use of Ackermann's lemma; our technic is based on unskolemization, which is made easer thanks to the use of the basic predicates *Atleast* and *Atmost*.

As the proposed result does not resolve the same problems as Scan and DLS, it is difficult to compare the power of these technics. However, we note that there are some formulas (pertaining to our verification problems) that our technic can resolve while Scan and DLS cannot. For instance, let us consider the following formula:

$$\exists f.\ f(p) \neq f(q) \wedge f(p) \neq f(r)$$

Scan and DLS cannot handle neither this formula as it, nor the following equivalent one, where f is replaced by the binary predicate R_f.

$$
\begin{array}{ll}
\exists R_f. & ((\forall x.\ \exists y.\ R_f(x, y)) \wedge (\forall x\ y\ z.\ (R_f(x, y) \wedge R_f(x, z)) \Rightarrow (y = z))) \wedge \\
& (\forall u.\ \neg(R_f(p, u) \wedge R_f(q, u))) \wedge (\forall u.\ \neg(R_f(p, u) \wedge R_f(r, u)))
\end{array}
$$

Our elimination technic rewrites the formula into the first-order one:

$$
\begin{array}{ll}
(\exists y\ y'\ y''. & ((y \neq y') \wedge (y \neq y''))\wedge \\
& (\forall y'''.\ \exists f.\ ((y''' = r) \Rightarrow (f = y''))\wedge \\
& \qquad\qquad ((y''' = q) \Rightarrow (f = y'))\wedge \\
& \qquad\qquad ((y''' = p) \Rightarrow (f = y))))
\end{array}
$$

5 Domain reduction-based technics

5.1 Bounded image functions

In this section, we propose an elimination result based on an S1S encoding of a bounded-image function. Then, we show that the previous results can be obtained through this technic.

5.1.1 The elimination result

A function of which image cardinality is less than n is said n-bounded. We define it as `Atmost(n,f,`$\lambda x.\top$`)`, which will be abbreviated `Atmost(n,f)`. Note that [4]: $\text{Atmost}(n, f) = \mid \mathcal{I}m(f) \mid \leq n$.

Thanks to the following result, each occurrence of a n-bounded image function can be replaced by another function defined through the use of second order (p_i) and first order (y_i) variables only:

$$\begin{array}{l} \text{Atmost}(n, f) = \\ \quad \exists p_1, \ldots, p_{n-1}, y_1, \ldots, y_n.\ f = \lambda\ e.\ p_1(e) \rightarrow y_1 \mid \ldots \mid p_{n-1}(e) \rightarrow y_{n-1} \mid y_n \end{array}$$

For instance, suppose we have to prove the formula:

$$\forall f.\ \text{Atmost}(n, f) \Rightarrow P(f)$$

According to the preceding definition, it is equivalent to the following formula where the quantification over f has been replaced by quantifications over unary predicates:

$$\forall y_1 \ldots y_n, p_1, \ldots, p_{n-1}.\ P(\lambda\ e.\ p_1(e) \rightarrow y_1 \mid \ldots \mid p_{n-1}(e) \rightarrow y_{n-1} \mid y_n)$$

Then, provided the preceding formula is an S1S term, its validity can be automatically decided either on the natural domain (S1S) or on any finite domain (WS1S).

In some particular cases, we actually know the elements of the image of f. Such a knowledge can be used to reduce the number of quantified variables. More precisely, in this context, the preceding transformation can be expressed as follows:

$$\begin{array}{l} (\mathcal{I}m(f) = \{y_1, \ldots, y_n\}) = \\ \quad \exists p_1, \ldots, p_{n-1}.\ f = \lambda\ e.\ p_1(e) \rightarrow y_1 \mid \ldots \mid p_{n-1}(e) \rightarrow y_{n-1} \mid y_n \end{array}$$

Remark It is possible to reduce the number of predicate variables to $O(\log n)$. For instance, if $n = 4$, we have:

$$f = \lambda\ e.\ p_1(e) \wedge p_2(e) \rightarrow y_1 \mid p_1(e) \rightarrow y_2 \mid p_2(e) \rightarrow y_3 \mid y_4$$

The minimization of predicate variables is a very useful transformation for decision tools like Mona [HJJ+95].

5.1.2 Segmented domain functions

We specialize the previous result to functions of which domain can be paritionned as a finite set of intervals over which the function is constant. These functions are said to be *segmented* and are defined by the following formula which only introduces first order variables.

$$\text{Segmented}(f, y_0, \ldots, y_n) = \exists b_0, \ldots, b_{n-1}.f = \lambda x.x < b_0 \rightarrow y_0 \mid \ldots \mid x < b_{n-1} \rightarrow y_{n-1} \mid y_n$$

From a practical point of view, we can use such a definition to validate communication problems where the contents of the message queues are segmented, e.g. in the alternated bit protocol or more generally in window-based protocols.

[4] $\mathcal{I}m(f)$ is the image set of f: $\mathcal{I}m(f) = \{y \mid \exists x.\ y = f(x)\}$

5.1.3 Application to functions with significative values

In this section, we study a general framework for applying the two preceding transformations. Such a framework is motivated by the fact that usually the $n-$*bounded* or the $n-$*segmented* hypothesis on a function f is not verified over all its definition domain but over a restriction of it.

Given a constant k, we define the function `Restr` as follows:

$$\mathrm{Restr}(f, D) = \lambda x.\ \mathbf{if} D(x)\ \mathbf{then}\ f(x)\ \mathbf{else}\ f(k) \tag{7}$$

The next theorem (8) states a sufficient condition for transforming an initial formula with the pattern $\forall f.\ P(f) \Rightarrow Q(f)$ into a new one where the hypothesis is put forward: $\forall f.\ \mathcal{H}(f) \Rightarrow P(f) \Rightarrow Q(f)$. This transformation relies on the definition of a reduction function r. The sufficient condition expresses that the initial formula is invariant for the reduction and that the predicate P (of the initial formula) implies that the function f satisfies $\mathcal{H}$ on the reduced domain. Formally, this condition is stated as follows:

$$\begin{array}{lll}(\forall\ f. & & ((P(f) \Rightarrow Q(f)) = (P(f\ \circ\ r) \Rightarrow Q(f\ \circ\ r))) \\ & \wedge & (P(f) \Rightarrow \mathcal{H}(f\ \circ\ r)))\end{array} \tag{8}$$

If the predicate $P(f)$ can be expressed as the conjunction $\mathrm{Atmost}(n, f, D) \wedge P'(f)$, the definition of r and $\mathcal{H}$ given in (9) allows to simplify the sufficient condition (8) into (10). Thus, the initial formula can be automatically decided provided that we prove manually the formula (10).

$$r = \lambda x.\ \mathbf{if}\ D(x)\ \mathbf{then}\ x\ \mathbf{else}\ k \qquad \mathcal{H}(f) = \mathrm{Atmost}(n+1, f) \tag{9}$$

$$\begin{array}{l}(\forall\ f.\ \mathrm{Atmost}(n, f, D) \Rightarrow \\ \quad ((P'(f) \Rightarrow Q(f)) = (P'(\mathrm{Restr}(f, D)) \Rightarrow Q(\mathrm{Restr}(f, D)))))\end{array} \tag{10}$$

5.1.4 Applications

We have experimented this transformation on the Illinois cache coherency protocol and on the alternated bit protocol. Concerning the Illinois protocol, the reduction domain has been defined as the set of valid caches, the restriction to this set of the cache function takes exactly one value. Concerning the alternated bit protocol, the reduction domain has been defined as the interval between the reception and the emission indexes of the communication buffer. The restriction of the `Buffer` function to this set takes at most two values.

5.1.5 Comparison with the unskolemization-based technics

It is interesting to compare the transformation of $n-$ bounded functions with the previous ones concerning functions applied on (4.1) or compared with (4.2) a given set of variables. As it is shown by the results (11) and (12), in order to eliminate such functions, we could have used the elimination technic for n-bounded functions. Actually (11) and (12) give equivalent formulas where the quantification is over $n-$ *bounded* functions. However, the obtained formula would be more complex.

Functions applied on a given set of universal variables We show that a universal quantification over a unary function which is only applied to a given set of universal variables can be transformed into a quantification over $n+1-$ bounded functions, where n is the cardinality of the set of variables. Then, thanks to the result of §5.1, the quantification over the function can be replaced by a quantification over unary predicates. More formally, we have:

$$\begin{array}{c} \forall f x_1 \ldots x_n.\ P(x_1, f(x_1), \ldots, x_n, f(x_n)) \\ = \\ \forall f x_1 \ldots x_n.\ \mathrm{Atmost}(n+1, f) \Rightarrow P(x_1, f(x_1), \ldots, x_n, f(x_n)) \end{array}$$

Actually, given a function f and a tuple of variables $x_1, \ldots, x_n$ we define the function $\overline{f}$ by: $\overline{f}(x) = \textbf{if } x \in \{x_1, \ldots x_n\} \textbf{ then } f(x) \textbf{ else } k$. The function $\overline{f}$ is trivially $n+1-$ bounded and we have:

$$P(x_1, f(x_1), \ldots, x_n, f(x_n)) = P(x_1, \overline{f}(x_1), \ldots, x_n, \overline{f}(x_n)) \tag{11}$$

Then, given that $\overline{f}$ is $n+1-$ bounded, we get the wanted result.

Functions compared with a given set of universal variables We show that a universal quantification over a unary function which images are only used in comparisons with a given set of universal variables can be transformed into a quantification over $n+1-$ bounded functions, where n is the cardinality of the set of variables. Then, as previously, the quantification over the function can be replaced by a quantification over unary predicates. More formally, we have:

$$\begin{array}{c} \forall f\ y_1 \ldots y_n.\ P(y_1, \lambda x.\ f(x) = y_1, \ldots, y_n, \lambda x.\ f(x) = y_n) \\ = \\ \forall\ y_1 \ldots y_n. \mathrm{Atmost}(n+1, f) \Rightarrow P(y_1, \lambda x.\ f(x) = y_1, \ldots, y_n, \lambda x.\ f(x) = y_n) \end{array} \tag{12}$$

The proof is the same as the previous one, but with $\overline{f}$ defined by[5][6]:

$$\overline{f}(x) = \textbf{if } f(x) \in \{y_1, \ldots y_n\} \textbf{ then } f(x) \textbf{ else } \epsilon y.\ y \notin \{y_1, \ldots, y_n\}$$

5.2 Comparisons with the image of a finite set of absolute or relative values

In this section, we propose domain restriction results for formulas where functional terms are only compared (through equality). More precisely, atomic formulas containing the quantified functions have the following patterns:

- $f(x) = g(x - r)$ where f and g are universally quantified functions and r is bounded by a constant R.
- $f(x) = c$ where c is universally quantified.

At the programming level, these patterns correspond to the use of arrays of an opaque type with the same syntactic restrictions. Universally quantified data correspond to program variables.

[5] ϵ denotes the choice operator

[6] In the **else** case, at least $f(x) \notin \{y_1, \ldots, y_n\}$ and thus we can choose a unique element not in this set.

5.2.1 The main result

Let n be the number of universal functions and p the number of universal variables. We consider formulas with the following pattern:

$$\forall f_1 \dots f_n\ k_1 \dots k_p.\ P((\lambda x.\ f_i(x) = k_j)_{\substack{1 \leq i \leq n,\\ 1 \leq j \leq p}} (\lambda x.\ f_i(x) = f_j(x-r))_{\substack{1 \leq i,j \leq n\\ 0 \leq r < R}})$$

We show that the validity of such a formula in a domain of more than $n \times R + p$ elements is equivalent to the validity of the same formula in a domain of exactly $n \times R + p$ elements. Consequently, the validity of the formula in any domain is equivalent to the validity of the formula in any domain of at most $n \times R + p$ elements.

Sketch of the proof Given a domain of at least $n \times R + p$ elements, and D a subset of $n \times R + p$ elements, we show the following equivalence:

$$\begin{array}{c}\forall f_1 \dots f_n\ k_1 \dots k_p.\ P((\lambda x.\ f_i(x) = k_j)_{\substack{1 \leq i \leq n,\\ 1 \leq j \leq p}} (\lambda x.\ f_i(x) = f_j(x-r))_{\substack{1 \leq i,j \leq n\\ 0 \leq r < R}}) \\ = \\ \forall g_1 \dots g_n\ h_1 \dots h_p.\ \bigwedge_{1 \leq i \leq n} \mathcal{I}m(g_i) \subset D \wedge \bigwedge_{1 \leq i \leq p} h_i \in D \Rightarrow \\ P((\lambda x.\ g_i(x) = h_j)_{\substack{1 \leq i \leq n,\\ 1 \leq j \leq p}} (\lambda x.\ g_i(x) = g_j(x-r))_{\substack{1 \leq i,j \leq n\\ 0 \leq r < R}})\end{array}$$

The right implication being trivial, only remains the left implication. Given a family of functions f_i and a family of constants k_i, there exists a one to one function a from $\{k_1, \dots, k_n\}$ to D. Let $h_i = a(k_i)$. We inductively define a family of functions g_i of which image is in D satisfying the following properties:

$$\begin{array}{c}(g_i(x) = h_j) = (f_i(x) = k_j) \\ (g_i(x) = g_j(x-r)) = (f_i(x) = f_j(x-r))\end{array} \tag{13}$$

The g_i functions are defined as follows:

$$g_i(x) = \begin{cases} h_j \textbf{ if } f_i(x) = k_j \\ g_j(x) \textbf{ if } j < i \wedge f_i(x) = f_j(x) \\ g_j(x-r) \textbf{ if } x \geq r \wedge j \leq i \wedge 0 < r < R \wedge f_i(x) = f_j(x-r) \\ \epsilon y.\ y \in D - \bigcup_{j<i}\{g_j(x)\} - \bigcup_j g_j(]x-R, x[) - \{h_1, \dots, h_p\} \textbf{ otherwise} \end{cases}$$

Given the chosen cardinalities, the choice operators are well defined.

We show by induction over x and over the index of the family of functions that the g_i satisfy the properties (13). Furthermore, by construction, the image of g_i are in D.

Consequently, in order to validate such of formula, it is sufficient to take $n \times R + p$-bounded functions.

5.2.2 Optimization for independent functions

We restrict to comparisons between the image by the same function of two neighbour points, or between an universal variable and the image of some point. The corresponding pattern is the following:

$$\forall f_1 \dots f_n\ k_1 \dots k_p.\ P((\lambda x.\ f_i(x) = k_j)_{\substack{1 \leq i \leq n,\\ 1 \leq j \leq p}} (\lambda x.\ f_i(x) = f_i(x-r))_{\substack{1 \leq i \leq n\\ 1 \leq r < R}})$$

In this case the size of the reduced domain can be set to $R+p$. For this purpose we take the family of functions g_i defined as follows:

$$g_i(x) = \begin{cases} h_j \textbf{ if } f_i(x) = k_j \\ g_i(x-r) \textbf{ if } x \geq r \wedge 0 < r < R \wedge f_i(x) = f_i(x-r) \\ \epsilon y.\ y \in D - g_i(]x-R, x[) - \{h_1, \ldots, h_p\} \textbf{ otherwise} \end{cases}$$

This formula is well defined if the domain D has at least $R+p$ elements.

5.3 Comparison with Wolper

One of the pioneering work on the use of automatic program verification for parameterized programs is that of Wolper [Wol86]. His work is stated in the framework of reactive systems modeled by non terminating programs that input and output data without modifying it. The semantics of a behaviour of a program is an infinite sequence of input or output events. In order to specify such a behaviour Wolper uses an extension of propositional temporal logic (PTL): an infinitary version where a countable set of propositions are considered and countable conjunctions and disjunctions are allowed. The main result of Wolper can be stated as follows: *under an assumption of data independence, programs that are defined over an infinite domain can be reduced to a smaller one and checked automatically by propositional temporal logic decision procedures.* Wolper's technic consists in the transformation of a formula containing comparisons between temporal data and constants into a formula where the domain of temporal data is finite. When interpreted in higher order logic the properties considered by Wolper are in fact those of section 5.2.2, the argument of universal function being time.

6 Case studies

In this section, we illustrate the presented technics on problems we have encountered when validating memory coherency protocols. The first case study concerns the validation of atomic memory protocols [AB86]: we use the unskolemization technic. The second case study deals with the validation of an abstract model for sequential consistency [Lam79].

6.1 Validation of atomic memory protocols

In this section, first we propose a generic formalization of an atomic memory protocol and its properties. The unskolemization technic is applied to the formula expressing that the generic protocol satisfies its properties. By showing that the Illinois protocol [AB86] is an instance of the generic one, we get equivalent decidable formulas for its correctness.

6.1.1 The generic invariant

As in section 2.4, we suppose that the used data structure is represented by the triplet `(C,Ctr,M)`. We parameterize the invariant by three predicates: a cache coherency predicate `C_Coh`, a memory coherency predicate: `M_Coh` and a control coherency predicate : `Ctr_Coh`. Then, the invariant is expressed as the conjunction of three predicates:

```
inv_gen(C_Coh,M_Coh,Ctr_Coh,C,Ctr,M) =
  Cache_Inv(C_Coh,C,Ctr) ∧ M_Inv(M_Coh,C,Ctr,M) ∧ Ctr_Inv(Ctr_Coh,Ctr)
```

For instance the predicate Cache_Inv is defined as:

$$\text{Cache\_Inv}(\text{C\_Coh},\text{C},\text{Ctr}) = \\ \forall u_1 u_2.\ \text{C\_Coh}(\text{Ctr}, u_1) \wedge \text{C\_Coh}(\text{Ctr}, u_2) \Rightarrow (C(u_1) = C(u_2))$$

6.1.2 The generic transition

In a similar way, we have three relations for constraining the new values of the cache, the memory and the control after a transition. A crucial point of the expression of these relations concerns the semantics of "data-independent" statements: it does not depend on a given value in the memory or in a cache. The expression of the generic transition is given in appendix C.

6.1.3 Unskolemization and instantiation

The unskolemization technic has been successfully applied to the formula expressing that the generic invariant is preserved by the generic transition. In the appendix, we give the text of the initial and transformed formulas. By showing that the invariant of the Illinois protocol is an instance of the generic one as well as each transition of this protocol is an instance of the generic one, all the formulas expressing the Illinois invariant correctness can be unskolemized. Since the only data function has been eliminated, a tool like Mona [HJJ+95] can be used to decide them automatically. (In the appendix, we give the expression of a read transition as a generic one (appendix D) as well as the expression of the Illinois invariant as a generic one (appendix E)).

6.2 Validation of an abstract sequentially consistent protocol

In order to validate the consistency model of the M3S protocol [SMRL92], we have exhibited an abstract model defined as a transition system [BCF97].

6.2.1 Specification of the model

The state space of this transition system consists in a sequence of memory operations, i.e., either a read or a write with a value. In such a sequence, memory operations can be inserted at different places. However, the atomicity, i.e., the property that any read value must be equal to the last written value, must be preserved. More formally, if we characterize the sequence of memory operations by the following variables:

- `h_op` the sequence of reads or writes,
- `h_v` the sequence of values,
- `h_last` the index of the end of the sequence,

the atomicity property, written as follows, must be an invariant of the program:

```
⟨ ∧ i : i < h_last :: (h_op[i] = Read) ⇒ (h_v[i−1] = h_v[i]) ⟩
```

The program of the abstract model is described by the following code[7]:

[7] the data structure **array** int **of** T is used as a support for a sequence

```
type MemAccess = { Read , Write };
Program Seqm3s ;
type
  Word;
var
  h_op    :  array int  of MemAccess; --   operations sequence
  h_v     :  array int  of Word;      --   values sequence
  h_last  :  int;                     --   index of the last of the sequence
invariant
  ⟨ ∧ i : i < h_last :: (h_op[i] = Read) ⇒ (h_v[i−1] = h_v[i]) ⟩ { Atomicity }
initially
  h_last = 0
begin
   --   Read insertion in an empty sequence
  ⟨ ▯ v :: h_last = 0 → h_op[h_last],h_v[h_last],h_last:= Read, v, h_last+1 ⟩ { Read }
▯ --   Read insertion in front
  ⟨ ▯ v :: h_v[0] = v →
        h_op[0]:=Read
     || ⟨ || k : k < h_last :: h_op[k+1],h_v[k+1]:=h_op[k],h_v[k] ⟩
     || h_last:=h_last+1 ⟩ { Read }
▯ --   Read insertion in the middle or at the end
  ⟨ ▯ v,i :: i < h_last ∧ h_v[i] = v →
        h_op[i+1],h_v[i+1]:=Read,v
     || ⟨ || k : i < k ∧ k < h_last :: h_op[k+1],h_v[k+1]:=h_op[k],h_v[k] ⟩
     || h_last:=h_last+1 ⟩ { Read }
▯ --   Write insertion at the end
  ⟨ ▯ v :: h_op[h_last],h_v[h_last],h_last:= Write,v,h_last+1 ⟩ { Write }
end --   Seqm3s
```

where the considered transitions express that:

- in an empty sequence, any value can be read,
- in a non empty sequence, only values already in the sequence can be read. A new sequence is computed in order to preserve the atomic consistency property.
- a write can be added at the end of the sequence.

6.2.2 The proof obligations

In order to validate the invariant, we have to show that the following predicates are always satisfied (the text of these predicates is given in appendix G):

- the invariant is true initially:
- the invariant is preserved by the four transitions.

Through standard rewriting rules, the obligations concerning the initialization, the insertion in an empty sequence and the write transitions are easily proved.

However, it remains to show the two following formulas concerning the other insertions (in front or in the middle and at the end).

```
-- proof obligation for read insertion in front
∀ h_op h_v h_last.
 (∀i. i < h_last ⇒ (h_op(i) = Read) ⇒ (h_v(i - 1) = h_v(i))) ⇒
   (∀i. i ≠ 0 ∧ i ≠ 1 ⇒ i - 1 < h_last ⇒ (h_op(i - 1) = Read)
          ⇒ (h_v(i - 2) = h_v(i - 1)))
```

```
-- proof obligation for read insertion in the middle or at the end
∀ h_op h_v h_last i.
 (∀i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v(i - 1) = h_v(i))) ⇒
  i < h_last ⇒
   (∀i'. if ((i = i' - 2) ∧ i' ≠ 0 ∧ i' ≠ 1)
         then (i' - 1 < h_last ⇒ (h_op(i' - 1) = Read) ⇒
                 (h_v(i' - 2) = h_v(i' - 1)))
         else if (((i' ≠ 0 ∧ i' ≠ 1) ∧ i < i' - 2)
         then (i' - 1 < h_last ⇒ (h_op(i' - 1) = Read) ⇒
                 (h_v(i' - 2) = h_v(i' - 1)))
         else ((h_op(i') = Read) ⇒ (i = i' - 1) ∧ i' ≠ 0
                                ∨ (h_v(i' - 1) = h_v(i')))))
```

These two formulas use the functional variable `h_v` which ranges over the opaque type `Word`. This variable only occurs in terms `h_v(i) = h_v(j)` (underlined terms of the formulas) where the difference between `i` and `j` is bounded by one. Consequently, the result of section (5.2) applies : the validity of the two formulas for any type `Word` is equivalent to the validity of the formulas for types `Word` of at most two elements. The derived formulas can be decided using an automatic tool such as Mona [HJJ+95].

7 Conclusion

We have presented two ways for validating second-order formulas automatically. The first one is based on quantification elimination and the other one on domain reduction. We have implemented all of our elimination technics within the HOL interactive theorem prover [GM94] for the reduction of terms and the MONA tool for the decision of WS1S formulas [HJJ+95]. The experimentation of some examples has led us to the following conclusions:

- technics based on normalization should be applied very carefully: trying them blindly is too expensive.
- although it is well known that the complexity of the decision procedure of the S1S logic is not elementary, the use of the MONA tool seems "raisonnable". In fact, we should say that MONA is very fast [8] on even complex true formulas and also very fast[8] to overflow memory on even simple false formulas.

We now plan to carry on our study in two directions

- first, it should be interesting to experiment the presented technics on other concrete problems. It seems that our technics are sufficient to prove automatically atomic memory protocols. Now, we have in mind the study of memory coherency problems implementing sequential consistency.
- second, another point which deserves more consideration is the study of programming and specification constructs of which semantics can be expressed inside the extension of the S1S logic we have studied. It should allow rigorous developments and verifications of programs such that some tedious proofs are avoided.

[8]sur SUN ULTRA 1 140 Mhz 128MO

References

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, nov 1986.

[Ack68] W. Ackermann. *Solvable cases of the decision problem.* NORTH-HOLLAND, 1968.

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

[BCF97] J.-P. Bodeveix, D. Carrière, and M. Filali. A refinement-based validation of a cache coherence protocol. In *10th International Conference on Parallel and Distributed Computing Systems. New Orleans, Louisiana USA.* ISCA, 1997.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10E20 states and beyond. In *5th Symposium on Logic in Computer Science*, jun 1990.

[BF95] J.-P. Bodeveix and M. Filali. On the refinement of symmetric memory protocols. In *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 58–74. Springer-Verlag, sep 1995.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[DLS95] P. Doherty, W. Lukaszewicz, and A. Szalas. Computing circumpscription revisited: a reduction algorithm. *Journal of automated reasonning,* (x):1–42, 1995.

[GM94] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1994.

[GO92] Dov. Gabbay and Hans Jürgen. Ohlbach. Quantifier elimination in second-order predicate logic. Technical Report 94-231, MPI, jul 1992.

[HJJ+95] J.G. Henriksen, J.L. Jensen, M.S. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A.B. Sandholm. Mona: Monadic second-order logic in practice. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, Aarhus, may 1995.

[Hol91] G.J. Holzmann. *Design and validation of computer protocols.* Prentice Hall, 1991.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, sep 1979.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The Stanford temporal prover. Technical Report STAN-CS-TR-94-151, Stanford University, jul 1994.

[SMRL92] P. Sainrat, A. Mzoughi, C. Rochange, and D. Litaize. The design of the M3S project : a multiported shared memory multiprocessor. In *Supercomputing'92*, nov 1992.

[Tho90] W. Thomas. Automata on infinite objects. In J.v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–192. MIT Press, 1990.

[vdD96] J. van der Does. Lectures on quantifiers. In *http://turing.wins.uva.nl/~jvddoes/books/Qlectures.ps.Z*, aug 1996.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In ACM, editor, *ACM Symposium on Principles of Programming Languages*, pages 184–193, jan 1986.

A The Illinois protocol

```
type
 Proc;
 Word;
 MemAccess = { Read , Write };
 State = {Inv, Excl, Shared, Dirty};
Program Illinois implements MemAccess;
var
 Ch :  array Proc  of Word;
 Ctr :  array Proc  of State;
 M   : Word;
invariant
 ⟨ ∧ p, q :: Ctr[p] ≠ Inv ∧ Ctr[q] ≠ Inv ⇒ Ch[p] = Ch[q] ⟩ { I0 }
 ⟨ ∧ p ::  ((Ctr[p] = Excl) ∨ (Ctr[p] = Shared)) ⇒ (Ch[p] = M) ⟩ { I1 }
 ⟨ ∧ p, q :: Ctr[p] ∈ {Excl, Dirty} ∧ Ctr[q] ≠ Inv ⇒ p = q ⟩ { I2 }
initially
 ⟨ ∧ p :: Ctr[p] = Inv ⟩
begin
     --  Read Hit
    ⟨ [] p :: Ctr[p] ≠ Inv → skip ⟩
  [] --  Read Miss et une copie non modifiée existe
    ⟨ [] p, q :: Ctr[p] = Inv ∧ Ctr[q] ∈ {Shared,Excl} →
      ⟨ || r :: Ctr[r] := Shared  if (r = p) ∨ (Ctr[r] ≠ Inv) ⟩
      || Ch[p] := Ch[q] { Read } ⟩
  [] --  Read Miss et une copie modifiée existe
    ⟨ [] p, q :: Ctr[p] = Inv ∧ Ctr[q] = Dirty →
    Ctr[p], Ch[p], M := Shared, Ch[q], Ch[q]
     || ⟨ || r :: Ctr[r] := Shared  if Ctr[r] ≠ Inv ⟩  { Read } ⟩
  [] --  Read Miss et il n'y pas de copie
    ⟨ [] p ::⟨ ∧ q :: Ctr[q] = Inv ⟩ → Ctr[p],Ch[p] := Excl,M  { Read } ⟩
  [] --  Write Miss
    ⟨ [] p, v :: Ctr[p] = Inv →
           Ctr[p], Ch[p] := Dirty, v || ⟨ || r :: Ctr[r] := Inv  if r ≠ p  ⟩  { Write }⟩
  [] --  Write Hit et la copie locale est modifiée
    ⟨ [] p, v :: Ctr[p] = Dirty → Ch[p] := v { Write } ⟩
  [] --  Write Hit et la copie locale est exclusive
    ⟨ [] p, v :: Ctr[p] = Excl → Ctr[p], Ch[p] := Dirty,v { Write } ⟩
  [] --  Write Hit et la copie locale est partagée
    ⟨ [] p, v :: Ctr[p] = Shared →
          Ctr[p], Ch[p] := Dirty,v || ⟨ || r :: Ctr[r] := Inv  if r ≠ p ⟩
      { Write } ⟩
  [] --  flush et la copie locale est modifiée
    ⟨ [] p :: Ctr[p] = Dirty → Ctr[p], M := Inv, Ch[p] ⟩
  [] --  flush et la copie locale est partagée ou exclusive
    ⟨ [] p :: Ctr[p] ∈ {Shared, Excl} → Ctr[p] := Inv ⟩
end --  Illinois
```

B Generic Invariant

```
inv_cache P c ctr = ∀ u1 u2. ((P ctr u1) ∧ (P ctr u2))  ⇒  (c u1 = c u2)

inv_mem P c ctr m = ∀ u1.(P ctr u1)  ⇒  (c u1 = m)

inv_ctr P ctr = P ctr

inv_gen P Q R (c,ctr,m) =
    (inv_cache P c ctr ) ∧ (inv_mem Q c ctr m) ∧ (inv_ctr R ctr)
```

C Generic Transition

```
tr_gen(f_c,f_c1,f_ctr,f_m,f_m1,c,ctr,m) =
             ( λ  u. f_c ctr u (c (f_c1 ctr u)) m,
              f_ctr ctr,
              f_m ctr (c (f_m1 ctr)) m
             )
```

D Instantiation of the generic transition

We show that the transition u_IllinoisRead1 can be considered as an instantiation of the generic transition: we have:

```
u_IllinoisRead1 p =  λ st st'.
  (st' = tr_gen
    ( λ ctr u v m. (g_IllinoisRead1 (( ε x.T),ctr,( ε x.T))) => (p = u => m |
v)| v)
    ( λ ctr u. u)
    ( λ ctr. (g_IllinoisRead1 (( ε x.T),ctr,( ε x.T))) =>
             ( λ u. (p = u) => Excl | (ctr u))
           | ctr)
    ( λ ctr v m. m)
    ( λ ctr. p)
    st)
```

E Instantiation of the generic invariant by the Illinois invariant

```
 Illinois_Inv =
    inv_gen ( λ ctr p. ~(ctr p = Inv))
            ( λ ctr p. (ctr p = Excl)  ∨  (ctr p = Shared))
            ( λ ctr. ∀p q. ctr p IN Dirty; Excl ∧ ~(ctr q = Inv)  ⇒  (p = q))
```

F Generic validation

The following formula asserts that the generic invariant is preserved by the generic transition:

```
∀ f_m1 f_m f_ctr f_c1 f_c R Q P.
       invariant (inv_gen P Q R)
         (λ mem mem'. mem' = tr_gen f_c f_c1 f_ctr f_m f_m1 mem)
```

After rewriting using the definitions, we obtain the next formula where `C` is the function to be eliminated:

```
  ((∀C Ctr m.
        ((∀u1 u2. Ctr_Coh Ctr u1 ∧ Ctr_Coh Ctr u2 ⇒ (C u1 = C u2)) ∧
         (∀u1. M_coh Ctr u1 ⇒ (C u1 = m))) ∧
        C_Coh Ctr ⇒
        (∀u1 u2. Ctr_Coh (f_ctr Ctr) u1 ∧ Ctr_Coh (f_ctr Ctr) u2 ⇒
               (f_c Ctr u1 (C (f_c1 Ctr u1)) m =
                f_c Ctr u2 (C (f_c1 Ctr u2)) m))) ∧
      (∀C Ctr m.
        ((∀u1 u2. Ctr_Coh Ctr u1 ∧ Ctr_Coh Ctr u2 ⇒ (C u1 = C u2)) ∧
         (∀u1. M_coh Ctr u1 ⇒ (C u1 = m))) ∧
        C_Coh Ctr ⇒
        (∀u1. M_coh (f_ctr Ctr) u1 ⇒
          ((f_c Ctr u1 (C (f_c1 Ctr u1)) m) = f_m Ctr (C (f_m1 Ctr)) m)))) ∧
     (∀C  Ctr m.
       ((∀u1 u2. Ctr_Coh Ctr u1 ∧ Ctr_Coh Ctr u2 ⇒ (C u1 = C u2)) ∧
        (∀u1. M_coh Ctr u1 ⇒ (C u1 = m))) ∧
       C_Coh Ctr ⇒ C_Coh (f_ctr Ctr))
```

The application of the proposed unskolemization technic leads to the following WS1S formula:

```
((∀Ctr m u1 m' u2 i x x'. ∃y. ∀z.
        (¬((y = x) ⇒ (y = f_c1 Ctr u2) ∧ (i = z)) ∨
         ¬((y = x') ⇒
           (y = f_c1 Ctr u1) ∧ ¬(f_c Ctr u1 z m = f_c Ctr u2 i m))) ∨
        (¬(z = m') ∧ Ctr_Coh Ctr y ∨ M_coh Ctr y ∧ ¬(z = m)) ∨
        ¬(C_Coh Ctr) ∨ ¬(Ctr_Coh (f_ctr Ctr) u1) ∨ ¬(Ctr_Coh (f_ctr Ctr) u2)) ∧
   (∀Ctr m u1 m' i x x'. ∃y. ∀z.
        (¬((y = x) ⇒ (y = f_m1 Ctr) ∧ (i = z)) ∨
         ¬((y = x') ⇒
           (y = f_c1 Ctr u1) ∧ ¬(f_c Ctr u1 z m = f_m Ctr i m))) ∨
        (¬(z = m') ∧ Ctr_Coh Ctr y ∨ M_coh Ctr y ∧ ¬(z = m)) ∨
        ¬(C_Coh Ctr) ∨ ¬(M_coh (f_ctr Ctr) u1))) ∧
  (∀Ctr m m'. ∃y. ∀z.
       (¬(z = m') ∧ Ctr_Coh Ctr y ∨ M_coh Ctr y ∧ ¬(z = m)) ∨
       ¬(C_Coh Ctr) ∨ C_Coh (f_ctr Ctr))
```

The instantiation of this formula on the Illinois protocol has been validated by MONA.

G Proof obligations

The invariant is true initially

```
∀ h_op h_v h_last.
  (h_last = 0) ⇒ (∀i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v (i - 1) = h_v i))
```

Read insertion in an empty sequence

```
∀ h_op h_v h_last h_op' h_v' h_last' v.
     (∀ i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v (i - 1) = h_v i)) ∧
     (h_last = 0) ∧
     (∀ x. h_op' x = ((h_last = x) → Read | (h_op x))) ∧
     (∀ x. h_v' x = ((h_last = x) → v | (h_v x))) ∧
     (h_last' = h_last + 1) ⇒
     (∀ i. i < h_last' ⇒ (h_op' i = Read) ⇒ (h_v' (i - 1) = h_v' i))
```

Read insertion in front

```
∀ h_op h_v h_last h_op' h_v' h_last' v.
  (∀ i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v (i - 1) = h_v i)) ∧
  (h_v 0 = v) ∧
  (∀ x. h_op' x = ((0 = x) → Read | ((x - 1 < h_last) → (h_op (x - 1)) | (h_op x)))) ∧
  (∀ x. h_v' x = ((x ≠ 0 ∧ x - 1 < h_last) → (h_v (x - 1)) | (h_v x))) ∧
  (h_last' = h_last + 1) ⇒
    (∀ i. i < h_last' ⇒ (h_op' i = Read) ⇒ (h_v' (i - 1) = h_v' i))
```

Read insertion in the middle or at the end

```
∀ h_op h_v h_last h_op' h_v' h_last' v i.
  (∀ i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v (i - 1) = h_v i)) ∧
  (i < h_last ∧ (h_v i = v)) ∧
  (∀ x. h_op' x = (((i = x - 1) ∧ x ≠ 0) → Read
                 | ((x ≠ 0 ∧ i < x - 1 ∧ x - 1 < h_last) → (h_op (x - 1))
                 | (h_op x)))) ∧
  (∀ x. h_v' x = (((i = x - 1) ∧ x ≠ 0) → v
                 | ((x ≠ 0 ∧ i < x - 1 ∧ x - 1 < h_last) → (h_v (x - 1))
                 | (h_v x)))) ∧
  (h_last' = h_last + 1) ⇒
  (∀ i. i < h_last' ⇒ (h_op' i = Read) ⇒ (h_v' (i - 1) = h_v' i))
```

Write insertion at the end

```
∀h_op h_v h_last h_op' h_v' h_last' v.
    (∀ i. i < h_last ⇒ (h_op i = Read) ⇒ (h_v (i - 1) = h_v i)) ∧
    (∀ x. h_op' x = ((h_last = x) → Write | (h_op x))) ∧
    (∀ x. h_v' x = ((h_last = x) → v | (h_v x))) ∧
    (h_last' = h_last + 1) ⇒
    (∀ i. i < h_last' ⇒ (h_op' i = Read) ⇒ (h_v' (i - 1) = h_v' i))
```