

On the automatic validation of parameterized Unity programs

J.-P. Bodeveix, M. Filali
e-mail: {bodeveix,filali}@irit.fr
fax: (33) 5 61 55 68 47

IRIT-Université Paul Sabatier
118 Route de Narbonne
F-31062 Toulouse cédex
France

Abstract. We study the automation of the verification of Unity programs with infinite or parameterized state space. This paper presents methods allowing the transformation of some second-order formulas expressing invariants into equivalent formulas expressed in a weaker but decidable logic. Two technics are considered: quantifier elimination and reduction to finite domain.

1 Introduction

In the last few years, many program validation tools have been developed [12, 14]. 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 [7]. 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.

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 semantics level, or we are at the program semantics 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 [2, 4]. 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 from the Illinois cache coherency protocol.

```

type
  Proc; -- opaque processor type
  Word; -- opaque word type
  State = {Inv, Excl, Shared, Dirty}; -- states of a cache
declare
  C : array Proc of Word; -- processor copy of memory data
  Ctr : array Proc of State; -- (per processor) state of the cache
  M : Word; -- memory

```

The tuple representing the program state is typed as follows:

$$(\text{Proc} \rightarrow \text{Word}) \times (\text{Proc} \rightarrow \text{State}) \times \text{Word}$$

Thus, a variable is represented by an access function to the corresponding element of the tuple.

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 with partially defined statements (array assignments) as well as with non-deterministic statements (as TLA [13] actions). 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:

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

where \mathcal{V} is the set of program variables.

The general schema of a parallel statement is $S_0 \parallel \dots \parallel S_n$ where S_i are simple or parallel conditional assignments. We can reorder the statements into $A_{v_0} \parallel \dots \parallel A_{v_m}$ 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: $A_v \equiv v := e \text{ if } c$. Its semantics is expressed by the following relation between the global states s and s' :

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

- If v is an array variable, we have:

$$A_v \equiv \begin{array}{l} \langle \parallel q :: v[n_1(q)] := e_1(q) \text{ if } c_1(q) \rangle \\ \parallel \dots \\ \parallel \langle \parallel q :: v[n_k(q)] := e_k(q) \text{ if } c_k(q) \rangle \end{array}$$

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 is¹:

$$\begin{aligned} \text{action}_v(s, s') = \\ \forall x. v(s', x) = \text{if } (\exists q. x = n_1(q) \wedge c_1(q)) \text{ then } e_1(\varepsilon q. x = n_1(q) \wedge c_1(q), s) \\ \dots \\ \text{elseif } (\exists q. x = n_k(q) \wedge c_k(q)) \text{ then } e_k(\varepsilon q. x = n_k(q) \wedge c_k(q), s) \\ \text{else } v(s, x) \end{aligned}$$

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

Thus, the action predicate looks like:

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

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 the following, we illustrate the proposed technics on the invariant preservation.

¹ ε denotes the choice operator

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:

$$\forall v_1 \dots v_n. P(v_1, \dots, v_n) \Rightarrow Q(\varphi_1(v_1, \dots, v_n), \dots, \varphi_n(v_1, \dots, v_n)) \quad (1)$$

where the state space has been represented by the tuple (v_1, \dots, v_n) and a statement by a tuple $(\varphi_1, \dots, \varphi_n)$. 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 [15], 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 [3] is a cache coherency algorithm for shared memory multiprocessors. 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 With respect to the data structure given in section (2.1), the safety of the algorithm can be specified by the following invariant expressed as the conjunction of three basic properties:

invariant

$$\begin{aligned} \forall p \ q. (\text{Ctr}[p] \neq \text{Inv} \wedge \text{Ctr}[q] \neq \text{Inv}) &\Rightarrow (\text{C}[p] = \text{C}[q]) && \{\text{I0}\} \\ \forall p. (\text{Ctr}[p] \in \{\text{Excl}, \text{Shared}\}) &\Rightarrow (\text{C}[p] = \text{M}) && \{\text{I1}\} \\ \forall p \ q. (\text{Ctr}[p] \in \{\text{Excl}, \text{Dirty}\} \wedge \text{Ctr}[q] \neq \text{Inv}) &\Rightarrow (p = q) && \{\text{I2}\} \end{aligned}$$

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:

$$\langle \parallel p, q : \text{Ctr}[p] = \text{Inv} \wedge \text{Ctr}[q] \in \{\text{Shared}, \text{Excl}\} :: \\ \quad \langle \parallel r :: \text{Ctr}[r] := \text{Shared} \text{ if } (r = p) \vee (\text{Ctr}[r] \neq \text{Inv}) \rangle \\ \quad \parallel \text{C}[p] := \text{C}[q] \\ \rangle \rangle$$

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

$$\forall \underline{C} \text{ Ctr } M \text{ p } \underline{q}. \text{Inv}(\underline{C}, \text{Ctr}, M) \wedge \text{g_IllinoisRead}(\text{p}, \underline{q}, \underline{C}, \text{Ctr}, M) \Rightarrow$$

$$\text{Inv}(\lambda u. \text{if } \text{p} = u \text{ then } \underline{C}(\underline{q}) \text{ else } \underline{C}(u),$$

$$\lambda u. \text{if } (u = \text{p}) \vee (\text{Ctr}(u) \neq \text{Inv}) \text{ then } \text{Shared} \text{ else } \text{Ctr}(u),$$

$$M)$$

where Inv and g_IllinoisRead are defined as:

$$\text{Inv}(\underline{C}, \text{Ctr}, M) = \text{I0}(\underline{C}, \text{Ctr}, M) \wedge \text{I1}(\underline{C}, \text{Ctr}, M) \wedge \text{I2}(\underline{C}, \text{Ctr}, M)$$

$$\text{g_IllinoisRead}(\text{p}, \underline{q}, \underline{C}, \text{Ctr}, M) = (\text{Ctr}(\text{p}) = \text{Inv}) \wedge \text{Ctr}(\underline{q}) \in \{\text{Excl}, \text{Shared}\}$$

The main purpose of the proposed technics will be to eliminate the quantification over \underline{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 [6]. 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 either infinite or of parameterized size. 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, in the parameterized case, we want to carry a proof that should not only be valid for one problem, but for a whole family of problems; so in the statement of our initial problem we do not have a given 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. After such reductions, the obtained domains becoming finite, we can apply model checking technics to decide the transformed formulas.

3 Overview of quantifier elimination techniques

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).

3.1 Overview of Ackermann

The Ackermann's techniques are among the first that were developed to solve the elimination problem [1]. In higher order logic, these results can be stated as the two following theorems:

Theorem 1. *Let B be a predicate which contains only positive occurrences of φ , what is expressed as follows: $(\forall x. \varphi(x) \Rightarrow \psi(x)) \Rightarrow (B(\varphi) \Rightarrow B(\psi))$. 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:*

$$\begin{aligned} &\vdash \text{Ackermann-1 } (\exists\varphi. \forall x. (\varphi(x) \vee A(x, z)) \wedge B(\lambda x. \neg\varphi(x))) = B(\lambda x. A(x, z)) \\ &\vdash \text{Ackermann-2 } (\exists\varphi. \forall x. (\neg\varphi(x) \vee A(x, z)) \wedge B(\varphi)) = B(\lambda x. A(x, z)) \end{aligned}$$

3.2 Overview of Scan and DLS

The Scan algorithm [9] 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 C-resolvents and C-factors of the clausal form of the formula. If this step succeeds, we get a first-order formula equivalent to the initial one.

As Scan, the DLS algorithm [8] reduces second-order formulas to logically equivalent first-order formulas. Its basic idea is to isolate positive and negative occurrences of ϕ in order to get an equivalent formula as $\exists\bar{x}(\exists\phi(A_1(\phi) \wedge B_1(\phi))) \vee \dots \vee (\exists\phi(A_n(\phi) \wedge B_n(\phi)))$, where A_i (resp. B_i) contains only positive (resp. negative) occurrences of ϕ , and then to apply one of Ackermann's results.

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.

4 Unskolemization-based technique

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

4.1 Normalization step

In this step, we try to transform the preceding formula, using the same basic rewrites as DLS, into an equivalent formula such as: $\exists\bar{u}. \exists f. Q_1 \wedge \dots \wedge Q_n$, where the Q_i have one of the following patterns:

- P1** $\forall\bar{x}. R(f(g(\bar{x})))$ where g does not use any variable bound in R^2 .
- P2** $\forall xy. R(x) \vee R(y) \vee (f(x) = f(y))$
- P3** φ where the arguments of the occurrences of f only contain free variables with respect to φ .

² We should have: $(\forall\bar{x}. R(f(g(\bar{x})))) = (\lambda g. \forall\bar{x}. R(f(g(\bar{x}))))g$

Example: let us consider the following formula (obtained while applying the elimination process to a proof obligation of the Illinois protocol) where C is the function to be eliminated:

$$\begin{aligned} & \neg(\forall C \text{ Ctr } M. \\ & \quad ((\forall u1 \ u2. P(\text{Ctr}, u1) \wedge P(\text{Ctr}, u2) \Rightarrow (C(u1) = C(u2))) \wedge \\ & \quad (\forall u1. Q(\text{Ctr}, u1) \Rightarrow (C(u1) = M))) \wedge \\ & \quad R(\text{Ctr}) \Rightarrow \\ & \quad (\forall u1 \ u2. P(f_ctr(\text{Ctr}), u1) \wedge P(f_ctr(\text{Ctr}), u2) \Rightarrow \\ & \quad \quad (f_c(\text{Ctr}, u1, C(f_c1(\text{Ctr}, u1)), M) = \\ & \quad \quad \quad f_c(\text{Ctr}, u2, C(f_c1(\text{Ctr}, u2)), M)))) \end{aligned}$$

The normalization step produces the following equivalent formula:

$$\begin{aligned} & (\exists \text{ Ctr}. R(\text{Ctr}) \wedge & (4.1) \\ & (\exists M \ u1. P(f_ctr(\text{Ctr}), u1) \wedge \\ & (\exists u2. P(f_ctr(\text{Ctr}), u2) \wedge \\ & \quad (\underline{\exists C}. (\forall u3 \ u4. \neg P(\text{Ctr}, u3) \vee \neg P(\text{Ctr}, u4) \vee (C(u3) = C(u4))) \quad \mathbf{P2} \\ & \quad \wedge (\forall u5. \neg Q(\text{Ctr}, u5) \vee (C(u5) = M)) \quad \mathbf{P1} \\ & \quad \wedge (f_c(\text{Ctr}, u1, C(f_c1(\text{Ctr}, u1)), M) \neq \\ & \quad \quad f_c(\text{Ctr}, u2, C(f_c1(\text{Ctr}, u2)), M)))) \quad \mathbf{P3} \end{aligned}$$

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

4.2 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: $\exists \bar{v}_i \forall x. Q_i(\bar{v}_i, x, f(x))$.

Then, existentially quantified variables are lifted up and put ahead (some renaming may be necessary). The resulting formula looks like:

$$\exists \bar{v}_1 \dots \bar{v}_n \exists f. \forall x. Q_1(\bar{v}_1, x, f(x)) \wedge \dots \wedge \forall x. Q_n(\bar{v}_n, x, f(x))$$

*Rewriting rules for the patterns **P1**, **P2** and **P3*** We now describe the transformation rules for each pattern:

$$\begin{aligned} \mathbf{P1} \quad & \forall \bar{x}. R(f(g(\bar{x}))) = \forall \bar{x} \ y. (y = g(\bar{x})) \Rightarrow R(f(y)) \\ \mathbf{P2} \quad & \forall x \ y. R(x) \vee R(y) \vee (f(x) = f(y)) = \exists e. \forall x. R(x) \vee f(x) = e \\ \mathbf{P3} \quad & \varphi \end{aligned}$$

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

$$\varphi(f(t)) = (\exists y. (\forall x. (x = t) \Rightarrow (f(x) = y)) \wedge \varphi(y)) \quad (2)$$

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 (4.1). The following rewrites illustrate the application of **P3** rule, C being the function to be eliminated:

$$\begin{aligned}
& f_c(\text{Ctr}, u1, \underline{C(f_c1(\text{Ctr}, u1))}, M) \neq \\
& \quad f_c(\text{Ctr}, u2, \underline{C(f_c1(\text{Ctr}, u2))}, M) \\
= & \{ (2) \} \\
& \exists y1. \forall x1. (x1 = f_c1(\text{Ctr}, u1)) \Rightarrow (C(x1) = y1) \wedge \\
& \quad f_c(\text{Ctr}, u1, y1, M) \neq f_c(\text{Ctr}, u2, \underline{C(f_c1(\text{Ctr}, u2))}, M) \\
= & \{ (2) \} \\
& \exists y1. \forall x1. (x1 = f_c1(\text{Ctr}, u1)) \Rightarrow (C(x1) = y1) \wedge \\
& \quad \exists y2. \forall x2. (x2 = f_c1(\text{Ctr}, u2)) \Rightarrow (C(x2) = y2) \wedge \\
& \quad f_c(\text{Ctr}, u1, y1, M) \neq f_c(\text{Ctr}, u2, y2, M) \\
= & \{ \textit{Extraction of existential quantifiers} \} \\
& \exists y1 y2. \forall x1. (x1 = f_c1(\text{Ctr}, u1)) \Rightarrow (C(x1) = y1) \wedge \\
& \quad \forall x2. (x2 = f_c1(\text{Ctr}, u2)) \Rightarrow (C(x2) = y2) \wedge \\
& \quad f_c(\text{Ctr}, u1, y1, M) \neq f_c(\text{Ctr}, u2, y2, M)
\end{aligned}$$

4.3 Unskolemization step

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

$$\exists \overline{v}_i \exists f. \left(\begin{array}{l} \forall x. Q_1(\overline{v}_1, x, f(x)) \\ \wedge \dots \\ \wedge \forall x. Q_n(\overline{v}_n, x, f(x)) \end{array} \right) = \exists \overline{v}_i. \forall x. \exists y. \left(\begin{array}{l} Q_1(\overline{v}_1, x, y) \\ \wedge \dots \\ \wedge Q_n(\overline{v}_n, x, y) \end{array} \right)$$

4.4 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 formulated by "The image of the subset characterized by $\neg R$ has at most one element" which can be denoted: $\text{Atmost}(1, f, \neg R)$. In fact, we can generalize the **P2** case to $\text{Atmost}(n, f, D)$ and $\text{Atleast}(n, f, D)$. Actually, these two dual predicates admit an existential as well as an universal expression. Such a versatility makes easy the transformation of either a positive, or a negative occurrence into an existential formula which can be unskolemized. More precisely³:

³ $c \rightarrow p \mid q$ denotes the conditional expression if c then p else q .

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

Such properties could be defined through extensions of classical logic with appropriate quantifiers [16]. For instance, **P2** could be stated through the **atmost 1** quantifier.

4.5 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 technique is based on unskolemization, which is made easier 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 techniques. However, we note that there are some formulas (pertaining to our verification problems) that our technique 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 can handle neither this formula as it, nor the following equivalent one, where f is replaced by the binary predicate R_f .

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

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

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

5 Domain reduction-based techniques

First we propose an elimination result based on an S1S encoding of a bounded-image function. A function of which image cardinality is less than n is said n -bounded. We define it as $\text{Atmost}(n, \mathbf{f}, \lambda x. \top)$, which will be abbreviated as $\text{Atmost}(n, \mathbf{f})$. Note that ⁴: $\text{Atmost}(n, f) = |\mathcal{I}m(f)| \leq n$.

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

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:

$$\text{Atmost}(n, f) = \exists p_1, \dots, p_{n-1}, y_1, \dots, y_n. f = \lambda e. p_1(e) \rightarrow y_1 \mid \dots \mid p_{n-1}(e) \rightarrow y_{n-1} \mid y_n$$

For instance, suppose we have to prove $\forall f. \text{Atmost}(n, f) \Rightarrow P(f)$

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:

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

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:

$$(\mathcal{I}m(f) = \{y_1, \dots, y_n\}) = \exists p_1, \dots, p_{n-1}. f = \lambda e. p_1(e) \rightarrow y_1 \mid \dots \mid p_{n-1}(e) \rightarrow y_{n-1} \mid y_n$$

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

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

5.1 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 .
- $f(x) = k$ where k is universally quantified.

At the programming level, these patterns correspond to the use of arrays of an opaque type of which elements are compared or assigned according to the previous patterns..

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:

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

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 $\{k_1, \dots, k_p\}$ 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:

$$\begin{aligned} (g_i(x) = h_j) &= (f_i(x) = k_j) \\ (g_i(x) = g_j(x-r)) &= (f_i(x) = f_j(x-r)) \end{aligned} \tag{3}$$

The g_i functions are defined as follows:

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

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 (3). Furthermore, by construction, the image of g_i are in D .

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

Restriction to 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 & \text{if } f_i(x) = k_j \\ g_i(x - r) & \text{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, \dots, h_p\} & \text{otherwise} \end{cases}$$

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

5.2 Comparison with Wolper

One of the pioneering works on the use of automatic program verification for parameterized programs is that of Wolper [17]. Wolper's techniques 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 corresponding to the restriction to independent functions (section 5.1), the argument of universal functions being time.

6 Conclusion

We have presented two ways for validating Unity invariants automatically. The first one is based on quantification elimination and the other one on domain reduction. We have implemented our elimination technics within the HOL interactive theorem prover [10] for the reduction of terms and the MONA tool[11] for the decision of WS1S formulas. Moreover, we have applied these technics for the validation of memory coherency models [5].

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 techniques 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 Unity programs such that some tedious proofs are avoided.

References

1. W. Ackermann. *Solvable cases of the decision problem*. NORTH-HOLLAND, 1968.
2. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992.
3. 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.
4. 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.
5. J.-P. Bodeveix and M. Filali. Quantifier elimination technics for program validation. Technical Report IRIT/97-44-R, IRIT, nov 1997.
6. 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.
7. K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.
8. P. Doherty, W. Lukaszewicz, and A. Szalas. Computing circumscription revisited: a reduction algorithm. *Journal of automated reasoning*, (x):1–42, 1995.
9. Dov. Gabbay and Hans Jürgen. Ohlbach. Quantifier elimination in second-order predicate logic. Technical Report 94-231, MPI, jul 1992.
10. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1994.
11. 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.
12. G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1991.
13. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.
14. 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.
15. W. Thomas. Automata on infinite objects. In J.v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–192. MIT Press, 1990.
16. J. van der Does. Lectures on quantifiers. In <http://turing.wins.uva.nl/~jvddoes/books/Qlectures.ps.Z>, aug 1996.
17. 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.