Automatically Proving UNITY Safety Properties with Arrays and Quantifiers

Xavier Thirioux

ENSEEIHT-IRIT (FRANCE)
e-mail : thirioux@enseeiht.fr

Abstract. We address the general problem of automatically proving safety properties of reactive systems within the UNITY model. We take up a relational and set-based approach, and define some techniques to represent instructions and properties, allowing us to deal with arrays and quantification. An integration of these techniques into the OMEGA calculator, which we make use of with a significant example, now allows us to think of deep automation of non trivial theorem proving.

1 Introduction

The results set out in this paper stem from our works related to the DADA project (Distributed Algorithms Design Assistant) [CHP96] , whose aim is to help the design and proof of distributed reactive programs, i.e. programs concurrently executed within an environment, with which they share neither global memory nor a common clock.

In this context, this paper deals with automatic proofs of distributed programs described, as well as their properties, within the UNITY formalism.

We specifically focus upon axiomatic safety properties (i.e. Hoare triples). Indeed, we choose this approach because it allows us to translate these properties into first order logic formulas, extended with arithmetical operators for comparison between two integer terms, taken from a Presburger-like linear decidable arithmetic (without multiplication). Moreover, the *weakest preconditions* can effectively be computed for any UNITY instruction and any predicate.

As for liveness properties, the elementary ones regarding a single state transition are easily tractable (i.e. the *ensures* operator). Nevertheless, this approach is not powerful enough to cope with the *leads-to* operator, which is the transitive and disjunctive closure of *ensures*, involving in most cases tough induction steps that are to a large extent impossible to mechanize, whatever power of the proof assistant we use.

A first step in the development led to an implementation of a UNITY-based environment, allowing us to automatically check safety properties of a subset of the language. The main restrictions of this prototype were the lack of arrays and quantifiers, preventing us from expressing parameterized algorithms.

Therefore, in answer to this problem, we propose a new coding of instructions and properties with arrays and quantifiers, thus greatly increasing the expressive power of our environment, while keeping its fully mechanized proving process of axiomatic safety properties, through an algorithm resembling a semi-decision procedure. The relevant feature in our work is that we don't make use of uninterpreted function symbols to

represent arrays any longer [TP97], because adding these functions led to undecidable problems.

For this purpose, as we did in our prototype, we chose the OMEGA calculator because classical theorem provers, from OTTER to COQ [TP96], are not well suited to symbolic arithmetic calculations which naturally arise from considerations about reactive programs with integer counters for instance.

In the remainder, we show the concepts naturally related to this approach, as well as their implementation into the OMEGA tool. In the end, we handle an example program to highlight our process.

2 The Presburger Calculator OMEGA

To make things clearer, we briefly describe the main principles of this tool and the way we can simply encode quantifier and array free axiomatic safety properties.

2.1 Presentation

OMEGA [KMP+96] is not a theorem prover, but instead a calculator capable of deciding whether a set (or a relation) of linear arithmetical constraints has solutions or not, and exactly computing them all, in a symbolic way.

It handles sets and relations over tuples of integer variables. This calculator is based on Presburger arithmetic, a class of logical formulas built from affine constraints over integer variables. Such a formula follows the grammar :

$$\phi \ ::= \ \phi \wedge \phi \ \mid \ \phi \vee \phi \ \mid \ \neg\phi \ \mid \ \exists \ v : \phi \ \mid \ \forall \ v : \phi \ \mid \ e \ op \ e'$$

where *op* is among $=, \neq, <, >, \leq, \geq$ and e, e' are affine expressions on the variables.

A tuple set is defined as : $\{[u_1, u_2, \cdots, u_n] : \phi\}$ where ϕ is a Presburger formula. A tuple relation is of the form: $\{[u_1, u_2, \cdots, u_n] \rightarrow [v_1, v_2, \cdots, v_n] : \phi\}$.

OMEGA then provides a variety of set and relational operators to combine these sets and relations, such as *union*, *subset*, *join*, etc.

2.2 Checking Specifications

We translate preconditions and postconditions into sets of states, and the multiple assignment into a tuple relation.

To check an assertion like $\{P\}\ s\ \{Q\}$, one can use the notion of *strongest postcondition* (sp). For a statement s and a predicate P, $sp\,(s, P)$ is the *strongest* state formula that holds after having executed s from a state in which P holds. Therefore, a formula Q is a necessary postcondition of P with respect to s if and only if it is a *weaker* formula than $sp\,(s, P)$, that is :

$$\{P\}\ s\ \{Q\} \ \textit{iff} \ sp\,(s, P) \Rightarrow Q$$

For an assignment s of the form : $\begin{array}{rrrr} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & if \quad g_1 \\ \cdots & & \cdots & if \quad \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & if \quad g_p \end{array}$

we obtain : $\{P\}\ s\ \{Q\} \ \Leftrightarrow\ \bigwedge_{i=1}^{p} (sp\,(\alpha_i, P \wedge g_i) \Rightarrow Q)\ \bigwedge (P \bigwedge_{i=1}^{p} \neg g_i \Rightarrow Q)$

where α_i is the assignment $v_1, v_2, \cdots, v_n\ :=\ e_1^i, e_2^i, \cdots, e_n^i$.

The formula $sp\,(s, P) \Rightarrow Q$ is handled as the set relation $\mathcal{S}\,(\mathcal{P}) \subset \mathcal{Q}$ where $\mathcal{S}$ is the function from states to states corresponding to the statement s and $\mathcal{P}$ and $\mathcal{Q}$ are the sets of states for which the predicates (or state formulas) P and Q hold. This approach avoids an explicit computation of the *strongest postcondition* and allows to compute neither *weakest precondition* nor *strongest postcondition* at the DADA environment level [CHP96].

3 Handling Quantifiers and Arrays

3.1 Introduction

We found a uniform way to translate arrays and quantifiers on array subscripts into a simple relational framework, which we can encode both assignments and properties within. We have no bounds (but size and speed in practice), or on the number of quantifiers and array references to be used, either on the complexity of assignments (thus any depth of indirection is allowed).

Nevertheless, we cannot exactly encode arrays, but rather only tuples of array elements. Sets of such tuples are "loose", in the sense that some of these sets have no meaning if interpreted as arrays, whereas every array can be included into some corresponding sets of tuples. So we compulsory reason *ab absurdo* and try finding a counter example to $\neg P$ to prove the property P. Moreover, this kind of reasoning prevents us from guessing a witness for each existentially quantified property to be proved.

We are also limited to proofs with a constant number of basic inference rules, meaning for instance that induction is far out of range. For that matter, our process is fundamentally a kind of elaborate case analysis, which is yet the mostly used kind of reasoning when speaking of axiomatic safety properties.

The salient feature is that we don't work on sets of states, but rather on sets of tuples of pairs in the form $subscript, value$.

It can be demonstrated that, when coding array references in such a way, finding a missing tuple of subscripts in the set of tuples representing the property $\neg(P \wedge \bigcirc_s \neg Q)$ is sufficient to prove the Hoare triple $\{P\}\ s\ \{Q\}$.

The set of missing tuples can be computed as $\mathcal{P}roj(s) \Leftrightarrow \mathcal{P}roj(s \setminus P / \neg Q)$[1] where $\mathcal{P}roj$ projects each tuple of pairs $subscript, value$ to its canonically fitting tuple of subscripts (more detailed in Sect.3.5).

[1] "$-$", "$\setminus$" and "$/$" denote respectively set difference, restriction on the domain and on the range of a relation.

3.2 Encoding Properties

Safety properties can no longer be proved by set inclusion, as we did for scalar variables in the previous environment, because we work on sets of subscripts.

Firstly, we forbid any quantified variable to reference more than one array element. For instance, $\forall i.A[i] > A[i+1]$ becomes $\forall i.\forall j.j = i+1 \Rightarrow A[i] > A[j]$.

The properties we consider are assumed to be under prenex normal form, i.e. quantifiers are the most outer symbols. Furthermore, prenex formulas are skolemized, i.e. existentially quantified variables are considered as uninterpreted functions of the universally quantified variables that syntactically precede them.

Once we have normalized these properties, the principle we have implemented merely consists in adding a component $subscript, value$ to (the left of) the tuples for each quantifier/reference we meet.

Examples. Here i_0 and j_0 stand for the skolem constant and function corresponding to underlined variables.

$$\mathcal{S}(\exists \underline{i}.\forall j.A[i] > B[j]) = \{[i, ai, j, bj] \ : \ i = i_0 \Rightarrow ai > bj\}$$
$$\mathcal{S}(\forall i.\exists \underline{j}.A[i] > A[j]) = \{[i, ai, j, aj] \ : \ j = j_0(i) \Rightarrow ai > aj\}$$

Those examples above highlight the fact that a set coding a property is subscript-wise and array-wise *complete*, in the sense that each tuple of subscripts for a given array must belong to this set, even if this tuple doesn't provide any kind of knowledge about its components. For instance, still in the first (resp. second) example, we can't say a single word on the tuples whose component i (resp. j) is not equal to i_0 (resp. j_0). Similarly in the second example, the $[i, \_, j_0, \_]$ tuple gives a piece of information about the array, whereas the $[j_0, \_, i, \_]$ tuple, though present, doesn't give any.

To have *complete* sets is the best way to insure the OMEGA tool can automatically find counter examples by exhaustive case analysis when combining sets of properties with assignment relations.

3.3 Encoding Assignments

The instructions we take into account are exclusively assignments of array elements. These instructions are encoded as relations between sets of pairs[2], according to this principle : Each assigned reference corresponds to a pair in the image tuple, and each reference that is required to compute the above assignment corresponds to a pair in the source tuple.

We respectively denote any array reference in the current and next state as r and r'. Moreover, s typically stands for a subscript and v for a value.

For the equations coding the translation process from assignments to relations are undoubtedly intricate, we rely on the reader's perspicacity to figure out the general scheme from the specific (but significative) following examples.

[2] As usual, we mean pairs in the form $subscript, value$.

Example 1. If an assignment involves only one element of an array, one must specify that the other ones stay unchanged. For instance, if we only care about two arrays A and B, the assignment $A[i_0] := B[j_0]$ is mapped to the relation :

$$\{[i, ai, j, bj] \to [i, ai', j, bj'] \ : ((i = i_0 \wedge j = j_0) \Rightarrow ai' = bj) \\ \wedge\ (i \neq i_0 \Rightarrow ai' = ai) \wedge (bj' = bj)\}$$

One can quote that in the case when $i = i_0$ and $j \neq j_0$, we cannot deduce the value of ai' ($A'[i_0]$) from the piece of knowledge we have.

Of course, we are not always forced to describe exactly what happens to each array, especially if a safety property doesn't reference it.

An equivalent phrasing consists in distinguishing the subscripts i_0 and j_0 from the other ones, with the help of a set union :

$$\begin{array}{l} \{[i_0, ai, j_0, bj] \to [i_0, bj, j_0, bj]\} \\ \cup\ \{[i_0, ai, j, bj] \to [i_0, ai', j, bj] \ :\ j \neq j_0\} \\ \cup\ \{[i, ai, j, bj] \to [i, ai, j, bj] \ :\ i \neq i_0\} \end{array}$$

Example 2. The instruction $\langle || i : 0 < i < N :: A[i], B[i] := A[i+1], A[i] + B[i] \rangle$ is mapped to R :

$$\{[i+1, ai1, i, ai, i, bi] \to [i, ai', i, bi'] \ :\ 0 < i < N \wedge ai' = ai1 \wedge bi' = ai + bi\}$$

The very important point is that we need the relation coding an assignment to be subscript-wise and array-wise *total* and *surjective* to keep an exhaustive tool, like in Sect. 3.2. So, for a given array, the set of source (resp. image) tuples contains all possible subscripts within given ranges. But this doesn't mean that the relation itself must be complete. One doesn't need to relate each source subscript to each image subscript.

Unlike Sect. 3.2, information must be kept maximal, meaning that all the permutations of subscripts referencing a given array must be generated.

In our second example, we thus have to add the tuples $[i, \_, i+1, \_, i, \_]$ to the source tuples with the $perm$ relation that permutates its first two pairs of arguments (referencing the same array A). Eventually, the relation that faithfully represents our assignment is $R \cup R(perm)$.

3.4 Relating Properties to Assignments

Putting into practice the representation of properties and assignments gives birth to an obvious problem of arity, because properties don't necessarily refer to the same (number of) array elements as assignments do. If arities are not equal (so we can't compute $s \setminus P \,/\, \neg Q$), we must inject (mathematically speaking) tuples from properties and assignments into one another to make them fit. Shortly, a tuple that contains less references to a given array than the others will be injected into them. The techniques involved in that process only depend upon permutations and other classical, though tedious combinatorial arguments, for what we won't get into any further detail about that. We will simply focus on some examples.

3.5 Proving Properties with Respect to Assignments

Because the principle of the proof process consists in finding counter examples, relating properties to assignments must lead to such a case. Counter examples are given by tuples that have disappeared from the original relation (before we add properties). But a simple set difference doesn't totally suit our needs. Indeed, some tuples may be kept in the relation, i.e. may satisfy the constraints between their components, at the price of adding an extra hypothesis, let us call it H. As a matter of course, tuples whose constraints contain $\neg H$ have disappeared, but they are definitely not counter examples. To extract those very ones, which have disappeared without any help, we have to ignore possible extra hypothesis about values of references, by focusing exclusively on subscripts. This means we merely project the tuples on spaces of subscripts, thus erasing information about values (recall Sect.3.1).

Last, but not least, if the previously computed set S of counter examples is not empty, we have to prove that existentially quantified variables can be freely chosen within their respective domains. For instance, if i_0 is an existential variable ranging in domain D, we must check $\{i_0 \in D\} \subseteq S$.

Failing to find a counter example for each possible values of existential witnesses directly ruins the proof and the best we can do is to try to prove the negation of the considered safety property, for which we are not at all guaranteed to succeed either. Indeed, the drawback of our method is that it ressembles a half-decision procedure. Because we work on loose arrays (a class that strictly contains all real arrays), we actually are not sure to get all counter examples. In the case too few such examples are found, neither truth nor falsity of the property can be stated.

Example. From now on, we will ignore array bounds when they don't supply any kind of pertinent knowledge, for the sake of clarity and simplicity.

We wish to prove the axiomatic invariant stating that an array A is sorted, translated into :

$$\texttt{invariant}\ \forall\, i,j\ :\ 0 < i < N\ \wedge\ 0 < j \leq N :: \ j = i+1 \Rightarrow A[i] \geq A[j]$$

that becomes :

$$\begin{array}{rl} Sorted & := \{[i, ai, j, aj]\ :\ \cdots\ \wedge\ j = i+1 \Rightarrow ai \geq aj\ \} \\ UnSorted' & := \{[i, ai, j, aj]\ :\ \cdots\ \wedge\ (i \neq i_0\ \vee\ j \neq j_0)\ \} \\ & \cup\ \{[i_0, ai, j_0, aj]\ :\ \cdots\ \wedge\ j_0 = i_0 + 1\ \wedge\ ai < aj\ \} \end{array}$$

Taking this instruction :

$$\langle || i :\ 0 < i < N :: \ A[i] := A[i] + A[i+1]\ \rangle\ ||\ A[N] := 2A[N]$$

that becomes at first :

$$\begin{array}{l} R := \{[i, ai, i+1, ai1] \rightarrow [i, ai']\ :\ 0 < i \leq N \wedge ai' = ai + ai1\} \\ \quad \cup\ \{[N, ai, \_, \_] \rightarrow [N, ai']\ :\ ai' = 2ai\} \end{array}$$

The projection we use is defined on relations :

$$Proj \ := \ \{([i, \_, j, \_] \rightarrow [i', \_, j', \_]) \rightarrow ([i, j] \rightarrow [i', j'])\}$$

Then, defining some relations for combinatorial reasons :

$$\begin{array}{ll} Perm & := \{[i, ai, j, aj] \rightarrow [j, aj, i, ai]\} \\ Inj1 & := \{[i, ai] \rightarrow [i, ai, \_, \_]\} \\ Inj2 & := \{[j, aj] \rightarrow [\_, \_, j, aj]\} \\ \text{and} & \\ R\_sym & := R \ \cup \ R(Perm) \\ R\_inj & := Inj1(R\_sym) \ \cap \ Inj2(R\_sym) \end{array}$$

According to the process we described, we must check :

$$Proj(R\_inj) \ \Leftrightarrow Proj(R\_inj \setminus Sorted \ / \ UnSorted') \ \neq \ \emptyset$$

provided i_0 (and j_0) can freely range over $[1, \cdots, N \Leftrightarrow 1]$.

3.6 Translating into the OMEGA Calculator

Translating from subscripts set based representation into the OMEGA syntax doesn't raise any serious problem.

First of all, we have to encode $p \Rightarrow q$ as $\neg p \vee q$ because OMEGA lacks the implication operator. Similarly, OMEGA does only support integer variables, so if we want an integer variable v to contain a boolean value, we will conventionally take $True$ to be $v > 0$.

The only tedious point is that we cannot encode assignments as relations any more, due to the impossibility to translate the $Proj$ection operators we make use of. Anyway, this is not a dead-end because we are only to change relations to sets of appended tuples, in the form $source@image$.

For instance, the relation $R := \{[i, ai, j, bj] \rightarrow [i', ai'] \ : \ \cdots\}$ naturally becomes $S := \{[i, ai, j, bj, i', ai'] \ : \ \cdots\}$.

As for the existentially quantified variables, we translate their skolemized form into OMEGA's uninterpreted function symbols, without any restriction, but a possible harmless reordering of variables inside tuples. Indeed, arguments of such functions are necessarily prefixes of tuples, so that universally quantified variables must be located to the left inside tuples. OMEGA allows to refer to any prefix of a tuple, with the Set variable, which represents the whole current tuple. The length of a prefix is set by the arity of the function.

We can conclude by asserting that the OMEGA tool is well suited to the encoding we have built, and meets our expectation for an automatic proof process.

In the following section, we consider a classical algorithm in the realm of distributed systems, for which we show the OMEGA translation and proof process.

4 Applying our Approach to Distributed Systems

A description of a distributed (parallel) algorithm usually takes the form $\langle s \ : 0 < s \leq N \ :: \ P(s) \ \rangle$, with s standing for a site (or process) number. Because each site has a local state, we are brought to use arrays, whose subscripts are site numbers, and whose elements are local variables distributed on sites.

As an example, let us express a program that makes a token move along a ring of sites, to model the problem of mutual exclusion between N sites ($N \geq 2$). So we have three arrays to represent respectively for each site the three relevant "actions" : requesting for exclusion (req), owning the token (tok) and reaching exclusion ($excl$).

Program $Token$; a famous example...
var
$req, excl, tok$: **array**$[1..N]$ **of** $Boolean$;
initially
$tok[1] \wedge \forall s \ : \ 1 < s \leq N :: \neg tok[s]$
assign
– the behaviour of the sites
$\langle [\!] s : 0 < s \leq N ::$
$req[s] := true$ **if** $\neg req[s]$
$[\!]$ $excl[s] := true$ **if** $req[s] \wedge tok[s]$
$[\!]$ $excl[s], dem[s] := false, false$ **if** $excl[s]$
$\rangle$
– the moves of the token
$\langle [\!] s : 0 < s < N ::$
$tok[s], tok[s+1] := false, true$ **if** $tok[s] \wedge \neg excl[s]$ **(**$\mathcal{M}(s)$**)**
$\rangle$
$[\!]$ $tok[N], tok[1] := false, true$ **if** $tok[N] \wedge \neg excl[N]$ **(**$\mathcal{M}(N)$**)**
end $Token$.

As an example, we present the translation of the instruction $\mathcal{M}(s)$ that makes the token move along the ring from one site $s_0 \neq N$ to its neighbour. The following set is obtained, in which $\neg excl[s_0]$ has been removed, because we will only care about properties of the token. All subscripts are assumed to range over their respective domains :

$$
\begin{aligned}
Move := & \{[s1, v1, s2, v2, s1, v1] : s1 \neq s_0 \ \} \\
& \cup \{[s_0, v1, s2, v2, s_0, v1'] : (v1 > 0 \ \wedge \ v1' \leq 0) \ \vee \ (v1 \leq 0 \ \wedge \ v1' \leq 0) \ \} \\
& \cup \{[s_0, v1, s2, v2, s_0 + 1, v1'] : v1 > 0 \ \wedge \ v1' > 0 \ \} \\
& \cup \{[s_0, v1, s_0 + 1, v2, s_0 + 1, v1'] : v1 \leq 0 \ \wedge \ v1' = v2 \ \} \\
& \cup \{[s_0, v1, s2, v2, s_0 + 1, v1'] : v1 \leq 0 \ \wedge \ s2 \neq s_0 + 1 \ \}
\end{aligned}
$$

Note that $Move$ is a bit more complicated than previous examples because we must take into account its guard (moreover, just recall that for a boolean value v, truth is represented by $v > 0$).

The other instructions (but $\mathcal{M}(N)$) don't affect the token array, so we will ignore them, as far as we are solely concerned by the moves of the token.

4.1 Some Safety Properties

Some basic properties are uniqueness and permanence of the token. In our context, uniqueness of the token is equivalently stated as an axiomatic invariant :

$$\texttt{invariant}\ \ \forall s, s' \ :\ 0 < s, s' \leq N \wedge (tok[s] \wedge tok[s'] \Rightarrow s = s')$$

This invariant supplies two sets defining the initial and final state spaces :

$$\begin{array}{ll} Unique & := \{[s, v, s', v'] \ :\ 0 < s, s' \leq N \ \wedge\ (v \leq 0 \ \vee\ v' \leq 0 \ \vee\ s = s')\ \} \\ NotUnique' & := \{[s, v, s', v'] \ :\ 0 < s, s', s_1 \neq s_2 \leq N \ \wedge\ v > 0 \ \wedge\ v' > 0\ \} \end{array}$$

where s_1 and s_2 stand for two different site number constants, where tokens are supposed to be located (just recall that we reason *ab absurdo*).

First, $Move$ needs to be symmetrical with regard to its two initial references :

$$\begin{array}{ll} Perm & := \{[s1, v1, s2, v2, s1', v1'] \to [s2, v2, s1, v1, s1', v1']\} \\ Move\_sym & := Move \ \cup\ Move(Perm) \end{array}$$

Afterwards, as the raw translation of the instruction doesn't suit our requirement for two final references (to collate the final property with $Move$), we must add a fourth reference to tuples, using :

$$\begin{array}{ll} Inj\_1 & := \{[s1, v1, s2, v2, s1', v1'] \to [s1, v1, s2, v2, s1', v1', \_, \_]\} \\ Inj\_2 & := \{[s1, v1, s2, v2, s1', v1'] \to [s1, v1, s2, v2, \_, \_, s1', v1']\} \\ Move\_4 & := Inj\_1(Move\_sym) \ \cap\ Inj\_2(Move\_sym) \end{array}$$

Now, we relate the properties to the assignment :

$$\begin{array}{ll} InjP & := \{[s1, v1, s2, v2] \to [s1, v1, s2, v2, \_, \_, \_, \_]\} \\ InjP' & := \{[s1, v1, s2, v2] \to [\_, \_, \_, \_, s1, v1, s2, v2]\} \\ Move\_P & := Move\_4 \ \cap\ InjP(Unique) \ \cap\ InjP'(NotUnique') \end{array}$$

The subscripts extracting relation is :

$$Subscripts := \{[s1, \_, s2, \_, s1', \_, s2', \_] \to [s1, s2, s1', s2']\}$$

So we check non-emptiness of :

$$Counter\_Examples := Subscripts(Move) \ \Leftrightarrow Subscripts(Move\_P)$$

And if so, we perform this last test, to make sure that both s_1 and s_2 freely range over their domain :

$$\begin{array}{ll} Free\_s_1\_s_2 & := \{[\,] \ :\ 0 < s_1 \neq s_2 \leq N\} \\ Range & := \{[s1, s2, s1', s2'] \to [\,]\} \\ Free\_s_1\_s_2 & \subseteq\ Range(Counter\_Examples) \end{array}$$

As for the permanence of the token, it can be simply stated as :

$$\texttt{invariant}\ \ \exists s : 0 < s \leq N :: tok[s]$$

In our set-based framework, it gives :

$$\begin{array}{ll} Exists & := \{[s,v]\ :\ 0 < s, s_3 \leq N\ \wedge\ (s \neq s_3\ \vee\ v > 0)\ \} \\ NotExists' & := \{[s,v]\ :\ 0 < s \leq N\ \wedge\ v \leq 0\ \} \end{array}$$

where s_3 plays the same role as s_1 (or s_2) does. This time, the property has one reference, whereas the assignment has two initial references (and one final). To get over with this example[3], we show how to inject $Exists$ and $NotExists'$:

$$\begin{array}{ll} InjP1 & := \{[s1,v1] \rightarrow [s1,v1,\_,\_,\_,\_]\} \\ InjP2 & := \{[s1,v1] \rightarrow [\_,\_,s1,v1,\_,\_]\} \\ InjP' & := \{[s1,v1] \rightarrow [\_,\_,\_,\_,s1,v1]\} \\ Move\_P & := Move \cap InjP1(Exists) \cap InjP2(Exists) \cap InjP'(NotExists') \end{array}$$

4.2 Further Investigation

To give a better insight of what is really computed by OMEGA, we show this short $Counter\_Examples$ set, containing the sets of subscripts (and therefore references) that falsify $Exists \wedge \bigcirc_s NotExists'$:

$$\begin{array}{l} Counter\_Examples := \\ \quad \{[s_3,s,s_3] : 1 \leq s_0 < s_3 \leq N\ \wedge\ 1 \leq s \leq s_3\} \\ \cup\ \{[s_3,s,s_3] : 1 \leq s_3 < s_0 < N\ \wedge\ 1 \leq s \leq N\} \\ \cup\ \{[s_3,s,s_3] : 1 \leq s_0 < s_3 < s \leq N\} \\ \cup\ \{[s_3,s,s_3+1] : s_0 = s_3\ \wedge\ 1 \leq s \leq s_3 < N\} \\ \cup\ \{[s_3,s,s_3+1] : s_0 = s_3\ \wedge\ 1, s \Leftrightarrow 1 \leq s_3 \leq s\ \wedge\ s_3 < N\} \\ \cup\ \{[s_3,s,s_3+1] : s_0 = s_3\ \wedge\ 1 \leq s_3 \leq s \Leftrightarrow 2\ \wedge\ s \leq N\} \\ \cup\ \{[s_0,s_0+1,s_0+1] : 1, s_3 \Leftrightarrow 1 \leq s_0 \leq s_3\ \wedge\ s_0 < N\} \end{array}$$

It can be checked that s_3 ranges over $[1..N]$, that is its whole original domain.

Though not very user-friendly, this result is perfectly what we expected to find, that is to say that all the situations in which counter examples were to be encountered are actually represented in this automatically computed set :

- in the case when $s_3 = s_0$, i.e. when the token is on the current site s_0, it is expected to be found on the site $s_3 + 1$ after the execution of the assignment (the token has moved to its neighbour) ;
- in the case when $s_3 \neq s_0$, the token is expected to be found on site s_3 after the execution of the assignment (it has not moved).

The other sets computed along the proof process, for any of the two properties we examine, are far too big and intricate to give a good idea of what is going on inside the computer, this is mainly due both to the exhaustive encodings we chose and to the very much elaborate way OMEGA handles sets, leading most of the time to counter intuitive representations.

[3] The remaining computations of the counter examples and the domain of s_3 are similar to the previous case.

5 Conclusion

We have presented an original translation from instructions and properties involving quantifiers and arrays to a simple and uniform representation using tuples of variables. Its expressivity seems to be great and we have not, until now, encountered a program that could not be faithfully embedded in our formalism (obviously disregarding undecidable arithmetic expressions). Particularly, synchronous parallelism as well as non-deterministic choice are easily tractable.

We succeeded, thanks to the well fitting OMEGA tool, in putting into practice the encodings of those instructions and properties, leading to fully automatic proof processes, within a somewhat restricted framework, yet powerful enough to express and prove most of common axiomatic safety properties. Nevertheless, the translation process is not yet automatic, and is still handled by the user, the forthcoming task is thus to remove the user from the whole proof process.

Another possible extension is to mix scalar variables with arrays, inside the same tuple structure. This could be easily performed, all the more because a scalar variable can be seen as an array with a unique store.

For the time being, the main stumbling block is efficiency. We are in pratice limited to prove only twicely quantified properties, because OMEGA, though not too greedy in space, has a non-elementary exponential complexity in time.

Last but not least, a thorough comparison with other kinds of tools, theorem provers, calculators or whatever endowed with arithmetic decision procedures or at least powerful heuristics, such as PVS [SOR93] or MONA [HJJ+95] would be worthwhile.

References

[CHP96] M. Charpentier, A. El Hadri, and G. Padiou. Preuve Automatique dans un Environnement de Développement Unity. *T.S.I. Technique et Science Informatiques*, 15(1), January 1996.

[HJJ+95] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. MONA : monadic second-order logic in practice. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 131–145, Aarhus, Denmark, May 1995. BRICS Notes Series NS-95-2.

[KMP+96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 1.1.0*. Dept. of Computer Science and Institute for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742, November 1996.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. A tutorial on Specification and Verification Using PVS (Beta Release). Technical report, Computer Science Laboratory, SRI International,, Menlo Park, CA, March 1993.

[TP96] X. Thirioux and G. Padiou. Assistance à la preuve de programmes Unity à l'aide du démonstrateur Coq. In *Journées du GDR programmation*, Orléans, November 1996.

[TP97] X. Thirioux and G. Padiou. Une approche relationnelle de preuve dans le formalisme UNITY. *T.S.I. Technique et Science Informatiques*, October 1997. Submitted to.