Automated Verification of Szymanski's Algorithm[1]

E. Pascal Gribomont and Guy Zenner

University of Liège (Belgium)

Abstract. The algorithm for mutual exclusion proposed by B. Szymanski is an interesting challenge for verification methods and tools. Several full proofs have been described in the literature, but they seem to require lengthy interactive sessions with powerful theorem provers. As far as this algorithm makes use of only the most elementary facts of arithmetics, we conjectured that a simple, non-interactive proof should exist; this paper gives such a proof, describes its development and how an elementary tool has been used to complete the verification.

1 Introduction

Many kinds of hardware and software systems are now tentatively subjected to formal verification. Among the most interesting ones are the algorithms used to ensure mutual exclusion in distributed networks. This is due to both practical and scientific reasons. First, it is of foremost importance to guarantee that concurrently running processes do not interfere in a destructive way; in fact, most of the communication, cooperation and synchronization problems between concurrent processes can be seen as variants of the mutual exclusion problem: this problem therefore deserves much attention. Furthermore, algorithms that solve the mutual exclusion problem ... or are thought to solve it, tend to be rather short but subtle pieces of code. As a result, their formal verification is challenging but still within reach of tools whose practical efficiency decreases badly when the size of the code to be verified increases.

Szymanski's algorithm implements mutual exclusion in a distributed network. Each process owns only three boolean variables, meaning that computations take place within a finite state space; this algorithm can be verified in a fully automated way with model checking methods and tools. The only problem is that the number n of processes is arbitrary. The size of the state space grows exponentially with n, so verification based on model checking is realistic (and very efficient) for small values of n only. Anyway, a general proof, valid for every integer n, would be more useful and more elegant, but can not be devised purely within the model checking paradigm. In fact, Szymanski's algorithm involves

[1] *Correspondence to:*
P. Gribomont
Institut Montefiore
ULg, Sart-Tilman, B 28
B – 4000 Liège (Belgium)

Phone: +32 4 366 26 67
Fax: +32 4 366 29 84
e-mail: gribomont@montefiore.ulg.ac.be

each process accessing the state of all other processes. Statements' guards are quantified formulas, and quantified formulas also appear in any useful global invariant. Even if the quantification is bounded (by the number n of processes), using first-order logic seems mandatory.

Our purpose in the sequel of this paper is to show that a particular kind of tautology checking is not impaired by the exponentially growing size of the state space, even in the case of algorithms like Szymanski's, where quantifications in both the guards and the invariant are numerous.

A version of Szymanski's algorithm is introduced in Section 2, a propositional model is built in Section 3, the verification techniques are addressed in Section 4 and the computer-aided verification of the algorithm is described in Section 5. Section 6 outlines comparison with other approaches and gives a conclusion.

2 Szymanski's algorithm, a hybrid version

This algorithm has been first proposed in [28] but has been slightly modified several times. The version considered here is adapted from [22].

There are n concurrent processes, numbered from 1 to n, which switch endlessly between a non critical section and a critical section. The main safety requirement is that at most one process at a time can be within its critical section. (Other requirements, not investigated here, are absence of deadlock, responsiveness and pseudo-linear waiting.)

Each process executes the same code and has only reading access to the variables owned by the other processes. Process p owns three boolean variables a_p, s_p and w_p, which are initially false. An abstract version of the code repeatedly executed by process p is given in Figure 1.

It is convenient to introduce some terminology. Process p can be either in the anteroom (executing lines 1 to 3) or in the doorway (line 4) or in the waiting room (lines 5 to 8) or in the inner sanctum (lines 9 to 13). Now, this abstract, coarse-grained version (atomic version in [22, 23]) is interesting since it grasps the essence of the algorithm, without bothering with potential interference problems between processes; it is also easy to guess what the appropriate invariant for this algorithm should be. However, the abstract version should not be implemented as such, with each line being an atomic, uninterruptible piece of code : this would be very unefficient. In fact, each line involving all processes $q \neq p$ should be split into $n-1$ (or, for line 11, $p-1$) more elementary statements, to be executed independently or sequentially.

From the point of view of methods and tools, both versions are challenging. The abstract, coarse-grained one involves quantified guards, but the finer-grained version (called "molecular" in [22,23], where both versions are considered) contains more transitions, with a more complex invariant, and gives rise to a more complex proof.

Presenting both versions in a paper of acceptable size is not possible, but we can show how to cope with both kinds of problems by addressing a "hybrid"

1. NCS (Non Critical Section)
2. $a_p := \mathit{true}$
3. await $\forall j \leq n : \neg s_j$
4. $(w_p, s_p) := (\mathit{true}, \mathit{true})$
5. if $\exists j \leq n : a_j \wedge \neg w_j$
6. then $s_p := \mathit{false}$
7. await $\exists j \leq n : s_j \wedge \neg w_j$
8. $s_p := \mathit{true}$
9. $w_p := \mathit{false}$
10. await $\forall j \leq n : \neg w_j$
11. await $\forall j < p : \neg s_j$
12. CS (Critical Section)
13. $(a_p, s_p) := (\mathit{false}, \mathit{false})$

Fig. 1. Abstract version of Szymanski's algorithm

version. We will split only one quantified statement, which is the crucial "countdown" statement just before the critical section (line 11). A boolean array X_p is introduced to control this; $X_p[q]$ holds as soon as process p has checked $\neg s_q$. This has to be done for all $q < p$, so initially $X_p[q]$ holds only for all $q \geq p$ (denoted $X_p = I_p$); when the countdown is completed, $X_p[q]$ holds for all q (denoted $X_p = TR$).

It is also convenient to make explicit the various control points of the system; for each control point ℓ, there is a predicate *at* ℓ which is true when the execution is at control point ℓ. The transitions executed by process p in our hybrid version are listed in Figure 2. A transition $(p_i, G \rightarrow S, p_j)$ can be executed by process p when the control state is at p_i and the memory state satisfies the guard G, that is, when formula $(\mathit{at}\ p_i \wedge G)$ is true; the effect of the transition is to alter the memory state according to the assignment S and to transfer the control at point p_j.

The links between control points and program variables are expressed by assertions (1).

$$\begin{aligned} \mathit{at}\ p_{5,9..13} &\Leftrightarrow s_p\ , \\ \mathit{at}\ p_{3..13} &\Leftrightarrow a_p\ , \\ \mathit{at}\ p_{5..9} &\Leftrightarrow w_p\ . \end{aligned} \tag{1}$$

(The expression $\mathit{at}\ p_{5,9..13}$ stands for the formula $\mathit{at}\ p_5 \vee \mathit{at}\ p_9 \vee \mathit{at}\ p_{10} \vee \mathit{at}\ p_{11} \vee \mathit{at}\ p_{12} \vee \mathit{at}\ p_{13}$.)

The terminology introduced above allows for the following intuitive description [22] of the strategy used in the algorithm to grant passage from one "room" into another, and access to the critical section:

A_0 : When p is in the inner sanctum, the doorway is locked.

$$
\begin{array}{l}
(p_1, \mathit{true} \longrightarrow \mathrm{NCS}, p_2) \\
(p_2, \mathit{true} \longrightarrow a_p := \mathit{true}, p_3) \\
(p_3, \forall j \leq n : \neg s_j \longrightarrow \mathit{skip}, p_4) \\
(p_4, \mathit{true} \longrightarrow (w_p, s_p) := (\mathit{true}, \mathit{true}), p_5) \\
(p_5, \exists j \leq n : a_j \wedge \neg w_j \longrightarrow s_p := \mathit{false}, p_7) \\
(p_7, \exists j \leq n : s_j \wedge \neg w_j \longrightarrow \mathit{skip}, p_8) \\
(p_8, \mathit{true} \longrightarrow s_p := \mathit{true}, p_9) \\
(p_5, \neg\exists j \leq n : a_j \wedge \neg w_j \longrightarrow \mathit{skip}, p_9) \\
(p_9, \mathit{true} \longrightarrow w_p := \mathit{false}, p_{10}) \\
(p_{10}, \forall j \leq n : \neg w_j \longrightarrow \mathit{skip}, p_{11}) \\
(p_{11}, q < p \wedge \neg X_p[q] \wedge \neg s_q \longrightarrow X_p[q] := \mathit{true}, p_{11}) \\
(p_{11}, X_p = TR \longrightarrow X_p := I_p, p_{12}) \\
(p_{12}, \mathit{true} \longrightarrow \mathrm{CS}, p_{13}) \\
(p_{13}, \mathit{true} \longrightarrow (a_p, s_p) := \mathit{false}, p_1).
\end{array}
$$

Fig. 2. Hybrid version of Szymanski's algorithm

A_1 : When p is about to leave the waiting room,
some process has entered the inner sanctum.
A_2 : Once p is in the latter part of the inner sanctum,
the waiting room and the doorway are empty.
A_3 : If p is in its critical section, then no process with smaller index
can be in the doorway, waiting room or inner sanctum.

Four supplementary assertions (2) formalize this:

$$
\begin{array}{rl}
A_0 : & \mathit{at}\, p_{8..13} \Rightarrow \neg \mathit{at}\, q_4 \\
A_1 : & \mathit{at}\, p_8 \Rightarrow \exists k \leq n : \mathit{at}\, k_{10} \\
A_2 : & \mathit{at}\, p_{11..13} \Rightarrow \neg \mathit{at}\, q_{4..9} \\
A_3 : & ((\mathit{at}\, p_{11} \wedge X_p[q]) \vee \mathit{at}\, p_{12,13}) \wedge q < p \Rightarrow \neg \mathit{at}\, q_{4..13}
\end{array}
\qquad (2)
$$

The countdown for access to the critical section takes place at state p_{11}; the behaviour of array X_p is formalized in assertions (3).

$$
\begin{array}{rl}
q \geq p & \Rightarrow X_p[q]\,, \\
(q < p \wedge X_p[q]) & \Rightarrow \mathit{at}\, p_{11}\,.
\end{array}
\qquad (3)
$$

Assume now that $I(p, q)$ denotes the conjunction of all assertions (1,2,3). As an invariant, the formula $\forall p \forall q \neq p\, I(p, q)$ is a likely candidate. We observe it is true initially, when each process p is at p_1, with a_p, s_p, w_p being false, and $X_p = I_p$. Besides, assertion A_3 expresses that p and q cannot be both at the same time in their critical section: $\mathit{at}\, p_{12} \Rightarrow \neg \mathit{at}\, q_{12}$ is a logical consequence of this assertion.

3 A propositional model

3.1 Introduction

In order to prove that I is an invariant, we have to check that every transition τ respects formula I; this is written $\{I\}\tau\{I\}$. We rely on the classical rule: the triple

$$\{A\}\ \tau\ \{B\}$$

holds if and only if the formula

$$A\ \Rightarrow\ wp[\tau; B]$$

is valid. The operator wp (weakest precondition) takes two arguments, a transition τ and a set of states represented by a formula B. The associated value is the set of all states whose τ-successor satisfies B. The operator wp is easily computable and many tools exist which automate this computation (in the sequel, we use our tool CAVEAT [12]). This symbolic computation is based on the identity

$$wp[x := f(x,y)\,;\ B(x,y)]\ =\ B(f(x,y),y)\,. \tag{4}$$

The assertion $I \Rightarrow wp[\tau; I]$ is the verification condition associated with τ. It is easy to generate them in a fully automatic way, but less easy to check them for validity. In general, these assertions belong to first order logic, especially when the guards of the transitions are quantified formulas.

3.2 Elimination of explicit quantifiers

Earlier verification of Szymanski's algorithm have been completed with the assistance of an interactive theorem prover for first order logic [22, 23]. Our purpose now is to do it again, but without interactive work. This is possible if we rewrite the problem in the framework of propositional logic. The first step of this rewriting is the elimination of quantifiers. To achieve this, we introduce six auxiliary variables (5); these variables are counters, whose role is to record the number of processes for which some boolean expression is true.

$$\begin{array}{ll}
\#(s) & =_{def} |\{j : 1 \le j \le n \wedge s_j\}|\,; \\
\#(w) & =_{def} |\{j : 1 \le j \le n \wedge s_j\}|\,; \\
\#(s\overline{w}) & =_{def} |\{j : 1 \le j \le n \wedge s_j \wedge \neg w_j\}|\,; \\
\#(a\overline{w}) & =_{def} |\{j : 1 \le j \le n \wedge a_j \wedge \neg w_j\}|\,; \\
\#(X_p) & =_{def} |\{j : 1 \le j \le n \wedge X_p[j]\}|\,; \\
\#(10) & =_{def} |\{j : 1 \le j \le n \wedge at\ j_{10}\}|\,.
\end{array} \tag{5}$$

The range of all these variables is the set $\{0, \ldots, n\}$; for instance, $\#(s)$ is the number of processes whose s-variable is true. With these notations, quantifiers can be eliminated, according to Figure 3. The quantifier-free version of the program is given in Figure 4.

Comment. This translation is not difficult but should be completed with care. For instance, the statement $s_p := \mathit{false}$ is adequately translated into the statement

$\exists j \leq n : a_j \wedge \neg w_j$	$\#(a\overline{w}) \geq 1$
$\forall j \leq n : \neg w_j$	$\#(w) = 0$
$\exists q \leq n : at\ q_{10}$	$\#(10) \geq 1$
$\exists j \leq n : a_j \wedge \neg w_j$	$\#(a\overline{w}) \geq 1$
$\longrightarrow$	$\longrightarrow$
$s_p := false$	$(s_p, \#(s)) := (false, \#(s) - 1)$

Fig. 3. Eliminating quantification

$(s_p, \#(s)) := (false, \#(s) - 1)$ only if we already know that s_p is always true before the execution (which is a consequence of the invariant).

The invariant is translated in a similar way (6) and the behaviour of the auxiliary variables is formalized in supplementary assertions (7), added to the invariant.

$$\begin{aligned} at\ p_{8..13} &\Rightarrow \neg at\ q_4 \\ at\ p_8 &\Rightarrow \#(10) \geq 1 \\ at\ p_{11\ 13} &\Rightarrow \neg at\ q_{4..9} \\ ((at\ p_{11} \wedge X_p[q]) \vee at\ p_{12,13}) \wedge q < p) &\Rightarrow \neg at\ q_{4..13}\,. \end{aligned} \tag{6}$$

$$\begin{aligned} \#(s) &\geq 0 \\ \#(w) &\geq 0 \\ \#(s\overline{w}) &\geq 0 \\ \#(a\overline{w}) &\geq 0 \\ \#(10) &\geq 0 \\ at\ p_{3,4,10..13} &\Rightarrow \#(a\overline{w}) \geq 1 \\ at\ p_{10..13} &\Rightarrow \#(s\overline{w}) \geq 1 \\ \#(X_p) &\geq n - p + 1 \\ \neg at\ p_{11} &\Rightarrow \#(X_p) = n - p + 1 \\ at\ p_{10} &\Rightarrow \#(10) \geq 1\,. \end{aligned} \tag{7}$$

3.3 Elimination of implicit quantifiers

We have to prove the triple $\{\forall p \forall q \neq p\, I(p,q)\}\tau\{\forall p \forall q \neq p\, I(p,q)\}$ for each transition τ. Due to symmetry, we can restrict to transitions executed by a single fixed station p, so 14 transitions have to be considered. The "countdown" transition also refers to some process $q < p$, so we also select a second station $q < p$.

Besides, the assertions of the invariant refer to (at most) two distinct stations, say p' and q'. Instead of proving the original triple (in fact, 14 triples) we can prove the simpler triples

$$\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(p',q')\}\,.$$

These stations can be distinct from p and q, or not. We need therefore to evoke at most four distinct stations, say p, q, r and s. As argued in [12], our proof task

$$
\begin{aligned}
&(p_1, \mathit{true} \longrightarrow \mathit{skip}, p_2)\\
&(p_2, \mathit{true} \longrightarrow (a_p, \#(a\overline{w})) := (\mathit{true}, \#(a\overline{w}) + 1), p_3)\\
&(p_3, \#(s) = 0 \longrightarrow \mathit{skip}, p_4)\\
&(p_4, \mathit{true} \longrightarrow (w_p,\ s_p,\ \#(w),\ \#(s),\ \#(a\overline{w})) :=\\
&\qquad\qquad (\mathit{true}, \mathit{true}, \#(w) + 1, \#(s) + 1, \#(a\overline{w}) - 1),\ p_5)\\
&(p_5, \#(a\overline{w}) \geq 1 \longrightarrow (s_p, \#(s)) := (\mathit{false}, \#(s) - 1), p_7)\\
&(p_7, \#(s\overline{w}) \geq 1 \longrightarrow \mathit{skip}, p_8)\\
&(p_8, \mathit{true} \longrightarrow (s_p, \#(s)) := (\mathit{true}, \#(s) + 1), p_9)\\
&(p_5, \#(a\overline{w}) = 0 \longrightarrow \mathit{skip}, p_9)\\
&(p_9, \mathit{true} \longrightarrow (w_p,\ \#(10),\ \#(w),\ \#(a\overline{w}),\ \#(s\overline{w})) :=\\
&\qquad\qquad (\mathit{false}, \#(10) + 1, \#(w) - 1, \#(a\overline{w}) + 1, \#(s\overline{w}) + 1),\ p_{10})\\
&(p_{10}, \#(w) = 0 \longrightarrow \#(10) := \#(10) - 1, p_{11})\\
&(p_{11}, q < p \wedge \neg X_p[q] \wedge s_q \longrightarrow (X_p[q], \#(X_p)) := (\mathit{true}, \#(X_p) + 1), p_{11})\\
&(p_{11}, \#(X_p) = n \longrightarrow (X_p, \#(X_p)) := (I_p, n - p + 1), p_{12})\\
&(p_{12}, \mathit{true} \longrightarrow \mathit{skip}, p_{13})\\
&(p_{13}, \mathit{true} \longrightarrow (a_p,\ s_p,\ \#(s),\ \#(a\overline{w}),\ \#(s\overline{w})) :=\\
&\qquad\qquad (\mathit{false}, \mathit{false}, \#(s) - 1, \#(a\overline{w}) - 1, \#(s\overline{w}) - 1), p_1)\,.
\end{aligned}
$$

Fig. 4. Quantifier-free version of Szymanski's algorithm

can be reduced to checking the following triples:

$$
\begin{aligned}
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(p,q)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(q,p)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(p,r)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(r,p)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(s,q)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(q,s)\},\\
&\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(r,s)\},
\end{aligned}
$$

However, we have assumed $q < p$, but know nothing about the positions of r and s. In the example outlined in [12], that was irrelevant since neither the code nor the invariant involved comparison between these numbers. This is no longer true here, so we have to check every possibility. For instance,

$$\{\forall p \forall q \neq p\, I(p,q)\}\ \tau\ \{I(s,q)\}$$

splits into three subcases: $s < q < p$, $q < s < p$ and $q < p < s$; for the last triple, there are 12 subcases. Last, it seems safe to assume that, in the precondition $\forall p \forall q \neq p\, I(p,q)$, only the seven instances $I(p,q), \ldots, I(r,s)$ will be needed, so we replace the quantified precondition by the conjunction of these instances.

3.4 Generation of verification conditions

Automating wp-calculus is easy, at least when all statements and assertions are quantifier-free.[2] The tool CAVEAT was used to generate the verification conditions. These conditions still contain predicates, like $q < p$ and $\#(w) = 0$; these predicates will be viewed as atoms by the tautology checker.

3.5 Additional hypotheses

A full reduction of first-order theories to propositional calculus is obviously impossible. Information is lost if, for instance, the predicate $\{\#(w) = 0\}$ is viewed as an atomic proposition ("pseudo-atoms" are surrounded by curly braces). As a consequence, specific axioms are needed to restore this information; they will be used as additional hypotheses in the verification conditions. The axioms maintain the consistency between the original variables like w_p, and the pseudo-atoms like $\{\#(w) = 0\}$, and also between the pseudo-atoms themselves. For instance, the axiom

$$\neg(w_p \wedge \{\#(w) = 0\})$$

was used as an additional hypothesesis for verifying Szymanski's algorithm.

4 Propositional verification techniques

Supplementing the set of additional hypotheses seems a fair price to pay for reducing the first-order validity problem to tautology checking. However, the elimination of implicit quantifiers has strongly increased the size and the number of verification conditions. In fact, even state-of-the-art tautology checkers can barely cope with really big formulas. Fortunately, propositional verification conditions have a very particular structure. If the invariant I is the conjunction $a_1 \wedge \cdots \wedge a_n$, then the condition $I \Rightarrow wp[\tau; I]$ can be written as

$$(a_1 \wedge \cdots \wedge a_n) \Rightarrow (c_1 \wedge \cdots \wedge c_n), \tag{8}$$

where typically the members of the set of hypotheses $H = \{a_1, \ldots, a_n\}$ are small, while the size n of this set is big. For the propositional model of the hybrid version of Szymanski's algorithm, the average value of n is 130, but it is much higher for the finer-grained, molecular version.

It is often convenient to consider conclusions $c_1, \ldots, c_n$ one at a time, i.e., to split the verification condition (8) into the (conjunctive) set of formulas (9).

$$\begin{array}{l} (a_1 \wedge \cdots \wedge a_n) \Rightarrow c_1\,, \\ \ldots \quad \ldots \\ (a_1 \wedge \cdots \wedge a_n) \Rightarrow c_n\,. \end{array} \tag{9}$$

[2] One should be careful to deal with array assignments, though: the usual assignment rule (4) has to be adapted for arrays. However, this induces no trouble in the present framework.

The reason is as follows. When condition (8) happens to be valid, most of the hypotheses have a role in a validity proof, which is therefore lengthy and not easily constructed. However, experimentation shows that, most of the time, each formula of the list (8) admits a short proof, using only a small subset H_j of the big set $H = \{a_1, \ldots, a_n\}$ of hypotheses; it is more efficient to construct n small proofs than to construct a big one.[3]

In order to discover the relevant set H_j of hypotheses, needed to establish conclusion c_j (in fact, as small a superset as possible), we use a three-phased approach. The first phase interleaves simplification and elimination of provably irrelevant hypotheses. A hypothesis h is *provably irrelevant* with respect to a conclusion c if formulas $(h \wedge \varphi) \Rightarrow c$ and $\varphi \Rightarrow c$ are equivalent for every formula φ. Syntactic criteria for simplification and irrelevance detection are developed in [13]. Second, the remaining hypotheses are sorted in a list $h_1, h_2, \ldots$; the idea is that seemingly most relevant hypotheses appear before seemingly less relevant ones. This notion is investigated in [13] so only an elementary example is given here. If c is $c_1 \Rightarrow c_2$, then a hypothesis like $c_3 \Rightarrow c_2$ is potentially useful (say, if $c_1 \Rightarrow c_3$ can be established from other hypotheses) and will therefore appear early in the list. On the contrary, a hypothesis like $c_2 \Rightarrow c_4$ is provably irrelevant, and therefore will not appear at all. (See [13] for more details.)
Comment. The nth step of the sorting procedure select $h_n \in H \setminus \{h_1, \ldots, h_{n-1}\}$ as the most promising hypothesis with respect to the formula

$$(h_1 \wedge \cdots \wedge h_{n-1}) \Rightarrow c$$

and not with respect to c alone.

Last, a tautology checker is iteratively called. The first attempt is to validate the conclusion c; if it fails, $h_1 \Rightarrow c$ is tried, and so on either until a member of the sequence

$$c,\ h_1 \Rightarrow c,\ h_2 \Rightarrow (h_1 \Rightarrow c), \ldots$$

is recognized as valid, or until the set of hypotheses has been exhausted. In the former case, the verification condition is valid.
Comment. The only cause of incompleteness in CAVEAT is the database of additional axioms. If the tautology checker does not provide a positive answer, either the program is not correct with respect to the invariant, or the database of additional axioms is too weak.

5 Automated verification

The proof of correctness of the hybrid version of Szymanski's algorithm with respect to its invariant has been completed with the tool CAVEAT (for Computer Aided VErification And Transformation). The first version of the tool has been presented in [12], but the version used here is rather different.

[3] This is usually true for verification conditions, but not for arbitrary propositional formulas.

The first component of CAVEAT is a classical generator of verification conditions. The data file contains the code of the program and the invariant as stated in Section 2, with a slightly different syntax. For the example considered here, the program contains 14 transitions and the invariant, in the form $I(p,q) \wedge \cdots \wedge I(r,s)$, is the conjunction of 66 assertions (after removing repetitions, since some assertions appear both in $I(p,q)$ and $I(p,r)$, for instance). The output file is the list of 14 verification conditions, obtained by weakest-precondition calculus. These are boolean formulas whose atomic propositions are true propositions (like w_p), place predicates (like $at\ q_4$) and pseudo-atoms, like $\{p < q\}$ and $\{\#(s) - 1 \geq 0\}$.

The second component of CAVEAT works with two data files, containing respectively one of the verification conditions (produced by the first component), and the list of additional hypotheses. The component simply inserts this list in the list of hypotheses of the verification condition, and then performs the splitting described in paragraph 4. This splitting allows for further elementary simplifications, so many of the $66 * 14$ produced conditions reduce to true and vanish; CAVEAT has in fact produced 155 conditions; one of them, corresponding to transition

$$(p_5, \neg\exists j \leq n : a_j \wedge \neg w_j \longrightarrow skip, p_9)$$

and to assertion

$$at\ p_{8..13} \Rightarrow \neg at\ q_4$$

is investigated further in the appendix; it contains 145 hypotheses, and the conclusion is

$$\{\#(a\bar{w}) = 0\} \Rightarrow (\neg at\ q_4)\,.$$

The list of additional hypotheses is the same for all verification conditions. For now, they are produced by the user, either from scratch, or by adaptation of some available databases of standard additional hypotheses. In our case, four databases have been used, for shared boolean arrays, identity, comparison operators ($\leq$, $\geq$, ...) and increment/decrement (properties of unary functions $x \mapsto x+1$ and $x \mapsto x-1$. For instance, the database for shared boolean arrays (like a, s and w in our example) is

$$\{\#(b) = n\} \Rightarrow b_p\,,\ b_p \Rightarrow \{\#(b) \geq 1\}\,,$$

whereas

$$\{x \geq 1\} \Rightarrow \neg\{x = 0\}$$

can be found in another database. Further relevant facts are obtained by combining several databases, like

$$\{\#(b) \geq 1\} \Rightarrow \neg\{\#(b) = 0\}\,.$$

The data file used by CAVEAT in our example contains 84 facts; a sample is given in the appendix.

The third component of CAVEAT applies simplification and elimination rules, in order to suppress as many hypotheses as possible. For the aforementioned

example, 90 hypotheses have been eliminated (out of 145); the remaining 55 hypotheses are listed in the appendix.

The fourth component consists in sorting the list of potentially relevant hypotheses, and the last component is the tautology checking itself. Recall that the latter work iteratively. For our example, only two hypotheses (out of 55) were really needed to establish the conclusion; their rank in the sorted list were 2 and 4, so five iterations were needed to validate the condition. (More details are given in the appendix.)

6 Discussion, comparison and conclusion

Szymanski's algorithm has been frequently used as benchmark for computer-aided verification tools. Two earlier successful attempts are [22] and [23]. We were puzzled by the fact that the critical problem in them was associated with the validation of the verification conditions. Clearly, those conditions are consequences of few elementary mathematical facts; the only difficulty is their huge size and number.

In fact, state-of-the-art theorem provers like recent versions of NQTHM or OTTER have been successfully used to prove non trivial mathematical theorems, but such systems are not at their best with very long formulas (see e.g. [9,21, 24,27]). That is the reason why the first version of CAVEAT emphasized propositional reasoning instead of first-order reasoning [12,13]. However, good results were obtained only for "nearly propositional" concurrent systems. In particular, Szymanski's algorithm was outside the practical scope of CAVEAT. To overcome this worrying limitation, we have eliminated the need of true first-order reasoning from CAVEAT; the only link between first-order reasoning and CAVEAT is now the database of additional hypotheses. The database itself is purely propositional; first-order reasoning is required only for its construction and, for now, little help is available to the user. The propositional part of CAVEAT works very efficiently, due to the simplification rules and the elimination rules based on the idea of relevance [13]; we have seen that the size of the formula effectively validated by the tautology checker was far smaller than the verification condition itself, 4 hypotheses instead of 145 in our example. As a result, the validation time needed by CAVEAT is quite short. Naturally, some time has to be spent for the construction of the database of additional hypothesis, but there is an important difference : the user faces only very short formulas, whose validity is trivial, instead of lengthy verification conditions. Besides, using first-order ATP tools is not easy. For instance, the strength of OTTER relies noticeably on an appropriate choice of resolution strategies; if the user is not skilled in this domain, the strength becomes weakness. In this respect, the main advantage of our proof of Szymanski's algorithm is that it relies only on *wp-sp* calculus, propositional logic, and the (most elementary) mathematical facts used in Szymanski's algorithm itself.

Validation is not the only difficult step with the assertional approach : the construction of the invariant can also be a problem. For coarse-grained versions

of concurrent algorithms, the construction of the invariant is simply a formal traduction of the main idea underlying the algorithm. However, as explicitly stated in [22], and also demonstrated e.g. in [11, 19], the invariant of a reasonably fine-grained version is a complex formula, whose construction can be challenging. Much work has been devoted to invariant construction, adaptation and approximation (see e.g. [2, 3, 5, 8, 10]). Most techniques rely on fixpoint calculus, implemented as weakest-precondition and strongest-postcondition calculi. These are already present in CAVEAT, and a module for invariant adaptation is planned; its purpose will not be the design of an invariant from scratch, but the incremental adaptation of a coarse-grained invariant into a finer-grained one (see [11] for a detailed presentation of this technique).

Another approach for concurrent program verification is model checking. Its main advantage is that little extra-work is required from the user, not even writing an inductive invariant. (See e.g. [4, 6, 14, 17, 25] for presentation and examples.) The problem is, model checking requires a finite, bounded model of programs and properties. Basically, a "parametric" system involving some unspecified integer constant n (the number of processes, for instance) can be investigated fully automatically only if n has been given an explicit numeric value; besides, the verification time may grow exponentially with this value. In spite of several attempts to overcome this limitation, e.g. [16, 30], and the combinatorial explosion [15, 26, 29], we feel this will remain a serious drawback (see also [1]). On the other hand, if we request a formal verification only for some fixed small values of parameters like n, model checking is the most convenient approach.

The approach presented here has also an educational advantage, especially if several versions of the same concurrent program are considered in sequence, coarser-grained version before finer-grained ones. The tool allows the user to understand more easily why some guard or some assertion is needed, for instance; it is also useful to the program designer, who is able to determine how fine-grained his/her algorithm can be implemented without destroying its properties.

Several tools and methods have been proposed which combine the assertional method and model checking, for instance [17, 18, 22]. As far as tautology checking is an elementary kind of model checking, the new version of CAVEAT also belongs in this category.

References

1. K.R. Apt and D.C. Kozen, Limits for Automatic Program Verification, *Inform. Process. Letters* **22** (1986) 307-309.
2. S. Bensalem, Y. Lakhnech and H. Saidi, Powerful techniques for the automatic generation, *Lect. Notes in Comput. Sci.* (1996) 323-335.
3. N. Bjorner, A. Browne and Z. Manna, Automatic Generation of Invariants and Intermediate Assertions, *Lect. Notes in Comput. Sci.* **976** (1995) 589-623.
4. J.R. Burch et al., Symbolic Model Checking: 10^{20} States and Beyond, Proc. 5th. Symp. on Logic in Computer Science (1990) 428-439.

5. E. Clarke, Program invariants as fixed points, Proc. 18th IEEE Symp. on Foundations of Comput. Sci. (1977) 18-29.
6. E. Clarke, E. Emerson and A. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Trans. Programming Languages Syst.* **8** (1986) 244-263.
7. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).
8. P. Cousot and N. Halbwachs, Automatic Discovery of Linear Restraints Among Variables of a Program, Proc. 5th ACM Symp. on Principles of Programming Languages (1978) 84-96.
9. D.M. Goldschlag, Mechanically Verifying Concurrent programs with the Boyer-Moore prover, *IEEE Trans. on Software Engineering* **16** (1990) 1005-1023.
10. S. Graf and H. Saidi, Verifying invariants using theorem proving, *Lect. Notes in Comput. Sci.* **1102** (1996) 196-207.
11. E.P. Gribomont, Concurrency without toil: a systematic method for parallel program design, *Sci. Comput. Programming* **21** (1993) 1-56.
12. E.P. Gribomont and D. Rossetto, CAVEAT: technique and tool for Computer Aided VErification And Transformation, *Lect. Notes in Comput. Sci.* **939** (1995) 70-83.
13. E.P. Gribomont, Preprocessing for invariant validation, *Lect. Notes in Comput. Sci.* **1101** (1996) 256-270.
14. G. Holtzmann, An improved protocol reachability analysis technique, *Software, Practice, and Experience*, **18** (1988) (137-161)
15. C.N. Ip and D.L. Dill, Verifying Systems with Replicated Components in Murφ, *Lect. Notes in Comput. Sci.* **1102** (1996) 147-158.
16. B. Jonsson and L. Kempe, Verifying safety properties of a class of infinite-state distributed algorithms, *Lect. Notes in Comput. Sci.* **939** (1995) 42-53.
17. R.P. Kurshan and L. Lamport, Verification of a Multiplier : 64 Bits and Beyond, *Lect. Notes in Comput. Sci.* **697** (1993) 166-179.
18. D. Kapur and M. Subramanian, Mechanically Verifying a Family of Multiplier Circuits, *Lect. Notes in Comput. Sci.* **1102** (1996) 135-146.
19. L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm, *Sci. Comput. Programming* **2** (1983) 175-206.
20. K. Larsen, B. Steffen and C. Weise, Fisher's protocol revisited : a simple proof using modal constraints, Proc. 4th DIMACS Workshop on Verification and Control of Hybrid Systems. New Brunswick, New Jersey, 22-24 October, 1995.
21. W. McCune, OTTER 3.0 Reference manual and guide, Argonne National Laboratory, 1994.
22. Z. Manna et al., STeP : the Stanford Temporal Prover (Draft), Report No. STAN-CS-TR-94-1518, Stanford University, June 1994.
23. M. Nagayama and C. Talcott, An NQTHM Mechanization of Szymanski's algorithm, Report No. STAN-CS-91-1370, Stanford University, June 1991.
24. D.M. Russinoff, A Verification System for Concurrent Programs Based on the Boyer-Moore Prover, *Formal Aspects of Computing* **4** (1992) 597-611.
25. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
26. A. Parashkevov and J. Yantchev, Space Efficient Reachability Analysis Through Use of Pseudo-root States, *Lect. Notes in Comput. Sci.* **1217** (1997) 50-64.
27. D.M. Russinoff, A Mechanically Verified Incremental Garbage Collector, *Formal Aspects of Computing* **6** (1994) 359-390.
28. B. Szymanski, A simple solution to Lamport's concurrent programming problem with linear wait, Proc. 1988 Int. Conf. on Supercomputing Systems (1988) 621-626.

29. P. Wolper and D. Leroy, Reliable Hashing without Collision Detection, *Lect. Notes in Comput. Sci.* **697** (1993)
30. P. Wolper and V. Lovinfosse, Verifying Properties of large Sets of Processes with Network Invariants, *Lect. Notes in Comput. Sci.* **407** (1990) 68-80.

A A worked-out example

The verification condition considered here is $(I \wedge D) \Rightarrow wp[\tau; a]$ were I is the invariant, D is the database of additional hypotheses, τ is transition

$$(p_5, \neg\exists j \leq n : a_j \wedge \neg w_j \longrightarrow skip, p_9)$$

and a is the assertion

$$at\ p_{8..13} \Rightarrow \neg at\ q_4\,.$$

Comment. The syntax used here is nearly the same as in [22]; it is more readable than CAVEAT syntax, where for instance

$$at\ p_{8..10}$$

becomes

```
at 8[p] or at 9[p] or at 10[p].
```

In the sequel, everything has been translated back into the external syntax.

A.1 The database of additional facts

It contains a set of 84 mathematical facts. All of them are obviously valid; the critical point is not to omit any of them. Here is a small sample:

$$\begin{array}{l}\{\#(w) = 0\} \Rightarrow \neg w_p\,,\\ \{\#(a\bar{w}) \geq 0\} \Rightarrow \{\#(a\bar{w}) + 1 \geq 1\}\,,\\ \{\#(10) \geq 0\} \Rightarrow \{\#(10) - 1 \geq 0\}\,,\\ \{\#(X_p) = n\} \Rightarrow X_p[q])\,.\end{array}$$

A.2 Sorting hypotheses

The simplification / elimination phase strongly reduces the number of hypotheses in a verification condition. In our example, 55 hypotheses were maintained; here they are:

s_p
$s_q \Rightarrow at\, p_{5,9..13}$
$s_r \Rightarrow at\, r_{5,9}$
$s_s \Rightarrow (at\, s_{5,9,11,12,13}$
$at\, q_{3..13} \Rightarrow a_q$
$a_r \Rightarrow at\, r_{3,4,5,7,9}$
$a_s \Rightarrow at\, s_{3,4,5,7,9,11,12,13}$
$at\, q_{5..9} \Rightarrow w_q$
$at\, r_{5,7,9} \Rightarrow w_r$
$at\, s_{5,7,9} \Rightarrow w_s$
$at\, q_{3,4,10..13} \Rightarrow \{\#(a\bar{w}) \geq 1\}$
$at\, s_{3,4,11,12,13} \Rightarrow \{\#(a\bar{w}) \geq 1\}$
$((at\, s_{11,12,13} \Rightarrow \{\#(s\bar{w}) \geq 1\}$
$at\, r_9 \Rightarrow (\neg at\, q_4)$
$\neg at\, q_8$
$\neg at\, q_{11,12,13}$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow (\neg\{\#(a\bar{w}) = 0\})$
$X_q[q]$
$X_s[s]$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_p) \vee (\neg w_r))$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_r) \vee (\neg w_s))$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_q) \vee (\neg w_s))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_p \vee s_r)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_r \vee s_s)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_q \vee s_s)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_p) \vee (\neg w_r))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_s) \vee (\neg w_r))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_s) \vee (\neg w_q))$

$at\, p_{5,9..13} \Rightarrow s_q$
$at\, r_{5,9} \Rightarrow s_r$
$at\, s_{5,9,11,12,13} \Rightarrow s_s$
a_p
$at\, r_{3,4,5,7,9} \Rightarrow a_r$
$at\, s_{3,4,5,7,9,11,12,13} \Rightarrow a_s$
w_p
$w_q \Rightarrow at\, q_{5..9}$
$w_r \Rightarrow at\, r_{5,7,9}$
$w_s \Rightarrow at\, s_{5,7,9}$
$at\, r_{3,4} \Rightarrow \{\#(a\bar{w}) \geq 1\}$
$at\, q_{10..13} \Rightarrow \{\#(s\bar{w}) \geq 1\}$
$at\, q_{8..13} \Rightarrow (\neg at\, r_4)$
$at\, r_9 \Rightarrow (\neg at\, s_4)$
$\neg at\, q_{10}$
$at\, q_{11,12,13} \Rightarrow (\neg at\, r_{4,5,7,9})$
$X_p[p]$
$X_r[r]$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_p) \vee (\neg w_q))$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_r) \vee (\neg w_q))$
$\{\#(a\bar{w}) \geq 1\} \Rightarrow ((\neg w_p) \vee (\neg w_s))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_p \vee s_q)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_r \vee s_q)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow (s_p \vee s_s)$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_p) \vee (\neg w_q))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_r) \vee (\neg w_q))$
$\{\#(s\bar{w}) \geq 1\} \Rightarrow ((\neg w_s) \vee (\neg w_p))$

The conclusion is

$$\{\#(a\bar{w}) = 0\} \;\Rightarrow\; \neg at\, q_4\,.$$

The sorting procedure produces this

h_1 : $at\, q_{3..13} \Rightarrow a_q$
h_2 : $at\, q_{3,4,10..13} \Rightarrow \{\#(a\bar{w}) \geq 1\}$
h_3 : $at\, r_9 \Rightarrow (\neg at\, q_4)$
h_4 : $\{\#(a\bar{w}) \geq 1\} \Rightarrow (\neg\{\#(a\bar{w}) = 0\})$
h_5 : ...

Hypothesis h_1 seems promising: it could be used to establish a_q, which in turn would refute the antecedent of the conclusion, and therefore validate the conclusion. Hypothesis h_2 is interesting too, also as a helper for refuting the antecedent of the conclusion. Hypothesis h_3 could be used more directly, to establish the consequent of the conclusion. The next hypothesis h_4 allows to conclude. In fact,

$$(h_2 \wedge h_4) \;\Rightarrow\; c$$

is propositionally valid.