CAVEAT: technique and tool for Computer Aided VErification And Transformation

E. Pascal Gribomont and Didier Rossetto
Institut Montefiore, Université de Liège, Sart-Tilman B28,
B-4000 Liège (Belgium)
gribomon,rossetto@montefiore.ulg.ac.be
accepted for presentation to CAV'95 (LNCS)

Abstract. We describe CAVEAT, a technique and a tool (under development) for the stepwise design and verification of *nearly finite-state concurrent systems (NFCS)*. A concurrent system is nearly finite-state when most of its variables have a finite range (Booleans, bounded integers). The heart of CAVEAT is a tool for verifying invariants, i.e., inductive safety properties. The underlying method is classical : formula I is an invariant for system $\mathcal{S}$ if and only if some formula $\Phi_I =_{def} \{I\}\mathcal{S}\{I\}$ is valid. If $\mathcal{S}$ is an NFCS, the formula Φ_I contains only a small set of non-boolean variables. CAVEAT uses the *connection method* to extract from Φ_I a (small) set Ψ of *paths* (some kind of assertions) about the non-boolean variables; Φ_I is valid if and only if all paths contain *connections*, i.e., are inconsistent. For typical NFCS given with a correct invariant, the formula Φ_I is rather large (more than 100 lines) but Ψ is quite small (a dozen one-line formulas). The second part of CAVEAT (not implemented yet) supports an incremental development method that is fairly systematic, but has proved to be flexible enough in practice.

1 Introduction

From the theoretical point of view, formal methods are a rather satisfactory answer to the problem of unreliable software. However, from the practical point of view, these methods are nearly useless without appropriate tools.

It is well-known that fully automatic tools for general program design and/or verification can not exist, so we have to be satisfied with semi-automatic tools and/or restricted classes of programs.

The most classical approach to non-automatic program verification is the invariant method. Its principle is to reduce the correctness problem ("Is this program correct w.r.t. this specification ?") to the validity problem ("Is this formula a valid formula of classical first order logic ?"). Even when automation is not considered, the invariant method has two drawbacks: it is restricted to safety properties, and it is "creative" in the sense that the validation of a safety property implies the (non-trivial) design of an adequate invariant, that is, a stronger safety property which can be proved by induction. The first problem has been satisfactorily solved by the introduction of temporal logic; the second problem is dealt with in more or less satisfactory ways, and for more or less general classes of programs. The pragmatic view (and CAVEAT is / will be a pragmatic tool) is that formal methods become interesting when, first, testing methods *really* prove disappointing, second, reliability is *really* required, and third, programs are subtle and tricky even when they are not long. This is often the case for concurrent, distributed, reactive systems, and the problem of invariant construction seems especially important for such systems. CAVEAT is an attempt to automate an invariant-based stepwise design/verification method introduced in [13, 14, 16].

Most earlier approaches to (semi-)automatic program verification have been based on (semi-)automatic theorem proving, for classical logic and sometimes for temporal logic. A pragmatic drawback of theorem provers is that they are mostly interactive. Even if the prover really performs the biggest part of the verification task, the user has to oversee the whole verification process, and from time to time needs to interact with it. The problem lies with the rather poor ability of proving systems to extract from a large set of mostly elementary verification steps the small subset which is outside the scope of purely automatic tools. The success of the semi-automatic theorem proving approach depends on the skill of the user [11].

A more recent approach is restricted to finite-state systems. The principle is that both the finite-state system and the specification can be modelled by a formula of propositional temporal logic, or by some kind of automaton. As a result, system verification is decidable, for instance by model checking algorithms [8, 29]. Recent improvements in the performances of computer systems, and also in the search algorithms, have led to rather powerful tools. This induced attempts to extend these techniques to some classes of infinite-state systems, but only moderately successful results have been obtained until now [20, 30]. On the contrary, some severe theoretical restrictions to this approach have been obtained [1]. Besides, when a tested finite-state program is incorrect, the verification system gives little high-level insight about how the program should be corrected; similarly, the validation of a correct program gives little insight about how the program works and why it is correct.

Another promising track comes from recent improvements in tautology checking, especially the connection method (see [5, 31]) and the concept of (ordered) binary decision diagram (see [6, 26]). It is rather natural to wonder whether these techniques remain *practically* usable outside pure propositional logic. CAVEAT has evolved from some successful experiments in this area.

Section 2 introduces CAVEAT with a very elementary example and discusses the main choices we have made in the strategy of invariant verification. Section 3 accounts for a more significant experiment and demonstrates the usefulness of the approach in a restricted but important class of applications. It also presents an introduction to incremental design and verification. Section 4 is a brief comparison with related works.

2 The heart of CAVEAT: tautological reduction

2.1 Position of the problem

A formula I is an invariant of a concurrent system $\mathcal{S}$ if, in all computations, successors of states satisfying I also satisfy I. Hoare's axiom, or the liberal version of Dijkstra's weakest precondition calculus, reduces the problem of invariant verification to the purely logical problem of validity checking. With familiar notation (illustrated below), the formula to validate is $\Phi_I =_{def} (I \Rightarrow wlp[\mathcal{S}; I])$. The construction of Φ_I, when $\mathcal{S}$ and I are given, is (usually) straightforward. The validation, however, is not, since Φ_I is typically a rather large formula.

A formula J expresses a safety property of $\mathcal{S}$ with initial condition A if every state of every computation satisfies J. This holds if and only if an invariant I of $\mathcal{S}$ exists such that $(A \Rightarrow I)$ and $(I \Rightarrow J)$. The standard verification problem is to determine whether some system $\mathcal{S}$ with initial condition A satisfies the safety property (expressed by) J.

If $\mathcal{S}$ is a non-parametric finite-state system, the formulas A, J, I and Φ_I are propositional and full automation is possible. However, the construction of the invariant I is not a trivial task. Model checking is usually more effective here, since an explicit form of the invariant I is not needed; the model checker simply verifies that all accessible states satisfy J. (The set of accessible states determines the strongest invariant implied by A, often denoted $sin[A; \mathcal{S}]$.)

Pure model-checking does not apply if $\mathcal{S}$ is an infinite-state system. In this case, A, J, I and Φ_I are formulas of some first-order language (for instance, the language of number theory) and the verification problem becomes theoretically unsolvable even for a rather restricted class of programs. The invariant method still works, but is not easily turned into a reasonably efficient semi-automatic method.

There is, however, a large and interesting class of "borderline" cases, for which Φ_I is a large formula with only few occurrences of non-boolean variables. The method illustrated in the sequel seems very promising for this class. For the sake of simplicity, it is first introduced with the help of a purely finite-state example, even though it does not show its full potential in this case.

2.2 The connection method

The connection method can be viewed as an efficient implementation of the classical tableau method, used to determine whether a formula or a set of formulas has a model. The principle of the method is to reduce the initial formula into sets of literals, in such a way that the initial formula has no model if and only if each of the sets of literals contains a *connection*, i.e., a tuple of contradictory elements. In a purely propositional framework, only pairs like $\{p, \neg p\}$ are considered. In our framework, a connection is a bit more general; typical instances are $\{x > y,\ x = y,\ x < y\}$ and $\{at\ \ell_0,\ at\ \ell_1\}$, where ℓ_0 and ℓ_1 are distinct locations of the same process.

The connection method can be a powerful technique [31]; it is illustrated in the sequel of this section, first with a simplistic example.

The example is a two-process naive mutual exclusion algorithm, that has to be checked for mutual exclusion. The set of processes is $\{P, Q\}$. Each process contains three locations, identified by subscripts 0 (idle state), w (waiting state) and c (critical state), so $P = \{p_0, p_w, p_c\}$ and $Q = \{q_0, q_w, q_c\}$. There are two

Boolean variables inP and inQ. The set of transitions is

$$\begin{aligned}\mathcal{T} \;=\; \{&(p_0,\ inP := true\ ,\ p_w)\ ,\quad (q_0,\ inQ := true\ ,\ q_w)\ ,\\ &(p_w\ ,\ \neg inQ \longrightarrow skip\ ,\ p_c)\,(q_w\ ,\ \neg inP \longrightarrow skip\ ,\ q_c)\ ,\\ &(p_c,\ inP := false\ ,\ p_0)\ ,\quad (q_c,\ inQ := false\ ,\ q_0)\ \}\ .\end{aligned}$$

Comment. The formal notation used here and in CAVEAT to write programs has been introduced in [14]. It is similar in spirit to many other notations based on states and transitions, e.g. the language of Action Systems introduced in [3].

There are two Boolean variables and each process has three control locations, leading to a state space of 36 possible states. The main safety property of interest is mutual exclusion, formalized as

$$J \;=_{def}\; \neg(at\ p_c\ \wedge\ at\ q_c)\ .$$

An acceptable initial condition is:

$$A \;=_{def}\; (at\ p_0\ \wedge\ at\ q_0\ \wedge\ \neg inP\ \wedge\ \neg inQ)\ .$$

An appropriate invariant I is

$$\begin{aligned}&(at\ p_c\ \Rightarrow\ (\neg inQ\ \vee\ at\ q_w))\ \wedge\ (at\ q_c\ \Rightarrow\ (\neg inP\ \vee\ at\ p_w))\ \wedge\\ &(at\ p_0\ \equiv\ \neg inP)\ \wedge\ (at\ q_0\ \equiv\ \neg inQ)\ .\end{aligned}$$

One sees easily that both $A \Rightarrow I$ and $I \Rightarrow J$ holds,[1] and CAVEAT is used to check that I is really an invariant. It is sufficient to show that formula $(I \Rightarrow wlp[\tau; I])$ holds for each transition τ, and we consider here $\tau = (p_w\ ,\ \neg inQ \longrightarrow skip,\ p_c)$. The corresponding verification formula, say Φ, is obtained by wlp-calculus:

$$\begin{aligned}&(\neg\neg inP\ \wedge\ (at\ q_0 \Rightarrow \neg inQ)\ \wedge\ (\neg inQ \Rightarrow at\ q_0))\ \ \Rightarrow\\ &(\neg inQ\ \Rightarrow\ [(\neg inQ\ \vee\ at\ q_w)\ \wedge\ (at\ q_c \Rightarrow \neg inP)\ \wedge\ \neg\neg inP\ \ \wedge\\ &\qquad (at\ q_0 \Rightarrow \neg inQ)\ \wedge\ (\neg inQ \Rightarrow at\ q_0)])\ .\end{aligned}$$

With standard elementary techniques, Φ is reduced into two formulas, i.e.,

$$((at\ q_0 \Rightarrow \neg inQ)\ \wedge\ (\neg inQ \Rightarrow at\ q_0))\ \ \Rightarrow\ \ (\neg inQ\ \Rightarrow\ (\neg inQ\ \vee\ at\ q_w)) \qquad (1)$$

and

$$(\neg\neg inP\ \wedge\ (at\ q_0 \Rightarrow \neg inQ)\ \wedge\ (\neg inQ \Rightarrow at\ q_0))\ \Rightarrow\ (\neg inQ \Rightarrow (at\ q_c \Rightarrow \neg inP))\ .$$

Comment. The first one should have been

$$(\neg\neg inP \wedge (at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0)) \Rightarrow (\neg inQ \Rightarrow (\neg inQ \vee at\ q_w))$$

but inP occurred only once, and has therefore been replaced by its polarity $\mathbf{T}$, leading to formula (1). This transformation and some similar ones are automated in CAVEAT.[2]

The *subformula tableau* for formula (1) is given in Figure 1.

Each line of the subformula tableau corresponds to a node of the syntactic tree of the formula (the tree is traversed depthfirst). Let us consider formula (1). The *polarity* of the formula itself (root line a_1) is $\mathbf{F}$, meaning that our "goal" (hopefully unreachable) is to falsify the (hopefully valid) formula a_1. This formula is an implication; it will be false if and only if its antecedent a_2 is true and

[1] Recall that each process is at exactly one location at a time; this location rule is "wired in" CAVEAT.

[2] These simplifications are classical in resolution-based theorem provers; see e.g. [23].

	Polarity	Formula	P-type	S-type
a_1	**F**	$((at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0))$ $\Rightarrow$ $(\neg inQ \Rightarrow (\neg inQ \vee at\ q_w))$	α	–
a_2	**T**	$(at\ q_0 \Rightarrow \neg inQ) \wedge (\neg inQ \Rightarrow at\ q_0)$	α	α
a_3	**T**	$at\ q_0 \Rightarrow \neg inQ$	β	α
a_4	**F**	$at\ q_0$	–	β
a_5	**T**	$\neg inQ$	α	β
a_6	**F**	inQ	–	α
a_7	**T**	$\neg inQ \Rightarrow at\ q_0$	β	α
a_8	**F**	$\neg inQ$	α	β
a_9	**T**	inQ	–	α
a_{10}	**T**	$at\ q_0$	–	β
a_{11}	**F**	$\neg inQ \Rightarrow (\neg inQ \vee at\ q_w)$	α	α
a_{12}	**T**	$\neg inQ$	α	α
a_{13}	**F**	inQ	–	α
a_{14}	**F**	$(\neg inQ \vee at\ q_w)$	α	α
a_{15}	**F**	$\neg inQ$	α	α
a_{16}	**T**	inQ	–	α
a_{17}	**F**	$at\ q_w$	–	α

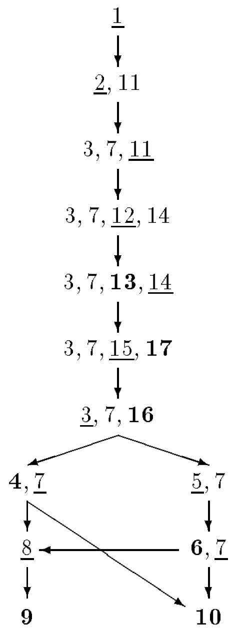
Fig. 1. Subformula tableau for formula (1)

its consequent a_{11} is false. As a result, the polarities of a_2 and a_{11} respectively are **T** and **F**. Besides, the P-type (*primary type*) of a_1 is α, meaning that a_1 is a conjunctive line.[3] The S-type (*secondary type*) of a line if the P-type of its father, so the root line has no S-type and the atomic lines (corresponding to atomic subformulas) have no P-type.

The subformula tableau is used to construct the *verification acyclic graph*, or *VAG*. The VAG corresponding to formula (1) is given in Figure 2 (left).

Each node in a VAG is a sequence of subformula indices; we distinguish *atomic* and *non-atomic* indices, corresponding respectively to atomic and non-atomic subformulas. The index i of an atomic subformula a_i is printed in boldface. The VAG is constructed from root to leaves according to the following rule. A node has successor(s) if it contains at least one non-atomic index i. If a_i has P-type α, there is only one successor, obtained by replacing i by j, k, where a_j and a_k are the immediate subcomponents of a_i (if a_i is a negation, there is only one subcomponent a_j). If a_i has P-type β, there are two successors, obtained

[3] Conjunctive lines are denied implications, denied disjunctions, asserted conjunctions; disjunctive lines (P-type is β) are asserted implications, asserted disjunctions and denied conjunctions. P-type is not really relevant for unary connectives, but we attribute P-type α to negations.



1	4, 9, 13, 16, 17	9, 13
2	4, 10, 13, 16, 17	4, 10
3	6, 9, 13, 16, 17	6, 9
4	6, 10, 13, 16, 17	6, 16

Fig. 2. Verification acyclic graph and path list for formula (1)

by replacing i by j and k, respectively. To save place, atomic indices of a node are not inherited by its successor(s). Leaves contain atomic indices only. The construction is non-deterministic, since a node can contain several non-atomic indices (in Figure 2, selected indices are underlined). The strategy of considering indices of P-type α first leads to a smaller VAG and is therefore adopted.

An example about formula (1). Three non-atomic indices 3, 7, 15 occur in node $(3, 7, 15, \mathbf{17})$; only index 15 is of P-type α, so it is selected. There is only one successor, obtained by replacing 15 by 16; besides, atomic index 17 is omitted.

It is now clear why the elementary claims introduced in paragraph 2.2 are called paths : each claim corresponds to a (maximal) path in the VAG. The last step of the connection method is to explore the VAGs and to list their paths. Each path connects the root of the VAG to a leaf and is identified in the list by the atomic indices occurring in the labels of its nodes. For instance, the first path of the VAG corresponding to formula (1) is

$$1 \rightarrow 2, 11 \rightarrow \ldots \rightarrow 3, 7, \mathbf{13}, 14 \rightarrow 3, 7, 15, \mathbf{17} \rightarrow 3, 7, \mathbf{16} \rightarrow \mathbf{4}, 7 \rightarrow 8 \rightarrow \mathbf{9}\,,$$

so this path will be identified as $\mathbf{4}, \mathbf{9}, \mathbf{13}, \mathbf{16}, \mathbf{17}$.

A formula is valid if each path of the corresponding path list contains a connection. The path list for formula (1) is given in Figure 2 (right), with a connection for each path. As a consequence, formula (1) is valid.

2.3 Concurrent construction and exploration of the VAG

An elementary but useful optimization consists in closing a path as soon as a connection is detected in it. This gives rise to shortened VAGs and path lists;

those for formula (1) are given in Figure 3.

$$
\begin{array}{c}
\underline{1} \\
\downarrow \\
\underline{2}, 11 \\
\downarrow \\
3, 7, \underline{11} \\
\downarrow \\
3, 7, \underline{12}, 14 \\
\downarrow \\
3, 7, \mathbf{13}, \underline{14} \\
\downarrow \\
3, 7, \underline{15}, \mathbf{17} \\
\downarrow \\
3, 7, \mathbf{16}
\end{array}
$$

1	3, 7, 13, 16, 17	13, 16

Fig. 3. Optimized VAG and shortened path list for formula (1)

2.4 The non-propositional case

The connection method reduces the problem of checking the validity of formula

$$\Phi \ =_{def} \ (a \wedge b \wedge c) \ \Rightarrow \ \neg(b \Rightarrow \neg a)$$

to the problem of finding a connection within each element of the path list

$$\Psi \ =_{def} \ (\{\mathbf{T} : a\,;\ \mathbf{T} : b;\ \mathbf{T} : c\,;\ \mathbf{F} : b\}\,,\ \{\mathbf{T} : a\,;\ \mathbf{T} : b;\ \mathbf{T} : c\,;\ \mathbf{F} : a\})\,.$$

The second problem is trivial, but the reduction process itself is not. Now, let us consider a rather similar case. The formula

$$\Phi' \ =_{def} \ (x > y \wedge b \wedge c) \ \Rightarrow \ \neg(b \Rightarrow x < y)$$

is valid if each path of the list

$$\Psi' \ =_{def} \ (\{\mathbf{T}\!: x > y\,;\ \mathbf{T}\!: b\,;\ \mathbf{T}\!: c;\ \mathbf{F}\!: b\}\,,\ \{\mathbf{T}\!: x > y\,;\ \mathbf{T}\!: b\,;\ \mathbf{T}\!: c;\ \mathbf{T}\!: x < y\})$$

is connected. The first path contains a trivial, propositional connection (atom b appears with both polarities) but the connection contained in the second path is $\{\mathbf{T} : x > y\,;\ \mathbf{T} : x < y\}$, which is non-propositional.[4]

The interesting point is that half the verification work (in this example) remains within the propositional framework and hence fully automatic. Our working hypothesis is that, for many "nearly finite-state systems", most of the invariant verification work will reduce to tautology checking and nearly all paths (say 99.9 %) will be closed (a connection will be found) in an automatic way. The main advantage we seek from our approach with respect to the more classical theorem-proving approaches, is the inherent ability of the connection method to "extract" the tiny fraction of the verification work which falls outside the propositional framework. This fraction is then isolated from the rest, and dealt

[4] More precisely, analysis of the "atoms" $x > y$ and $x < y$ is needed to detect that they are contradictory.

with in a classical way; either we use a knowledge base and a theorem prover that would tell us that $x > y$ and $x < y$ are not simultaneously satisfiable, or we simply report to the user the sublist of unconnected paths. The example presented in the next section illustrates what can be achieved even without ATP (automatic theorem proving).

3 Ricart and Agrawala's algorithm

It is now usual to verify some fine-grained version of a concurrent system by first considering some coarser-grained version(s). This approach was already used in [9] and [21] and is turned into a systematic method in [14] and [16]. We extract from the latter [16, p. 43] an intermediate, medium-grained version of Ricart and Agrawala's N-process mutual exclusion algorithm, introduced in [27].

3.1 The algorithm and its invariant

The basic idea, as introduced in [27], is as follows. A node attempting to invoke mutual exclusion sends a request to all other nodes. On receipt of the request, the other nodes send an immediate reply or defer it. When all replies have been received, the access to the critical section is granted. A deferred reply is delayed until the replying node has completed its own access to the critical section.

Some notation is introduced, mostly in accordance with [27].

rcs_p :	p needs access to the resource;
[ ] :	internal activity, able to alter rcs only;
RCS_p :	p requests access to the resource (Request Critical Section);
OR_p^q :	p waits for a reply from q (Outstanding Reply);
RD_p^q :	p defers a reply to q (Reply Deferred);
SN_p :	p has requested access at that time (Sequence Number) (time-stamp, implementing Lamport's "Bakery algorithm");
$SN_p < SN_q$:	p takes precedence over q;
$time$:	monotonically increasing integer, models real time;
$\mathcal{P}$:	the set of nodes; $\mathcal{P} = \{p, q, r, \ldots\}$;
N :	the number of nodes; $\lvert\mathcal{P}\rvert = N$;
$\mathcal{P}_p$:	short for $\mathcal{P}\backslash\{p\}$; $\lvert\mathcal{P}_p\rvert = N - 1$;
X_p :	variable ranging over subsets of $\mathcal{P}_p$.

The transitions of $\mathcal{S}$ are given in Figure 4 (for all distinct stations p and q).

In order to switch, say, from control location p_2 to p_3, node p has to send a request to each member of $\mathcal{P}_p$. In system $\mathcal{S}$, the $N-1$ corresponding messages are already modelled by $N - 1$ distinct transitions, but each of them remains rather abstract and the communication itself is modelled by switching the Boolean variable OR_p^q from false (i.e. 0) to true (i.e. 1), just as if communications always were reliable and timeless; system $\mathcal{S}$ is not very coarse-grained, but not really fine-grained either. Also note that the $N - 1$ communications are performed in arbitrary order; $q \in X_p$ means, "node q has been issued the request from p". Intuitively, the predicate $at^q\, p_3$ means: "from the point of view of station q, the

$(p_0,\ \neg rcs_p \ \longrightarrow\ [\,],\ p_0)\,,$
$(p_0,\ rcs_p \ \longrightarrow\ (RCS_p, SN_p, time) := (1, time, time+1)\,,\ p_2)\,,$
$(p_2,\ q \in \mathcal{P}_p \backslash X_p \ \longrightarrow\ (OR_p^q, X_p) := (1, X_p \cup \{q\})\,,\ p_2)\,,$
$(p_2,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_3)\,,$
$(p_3,\ q \in \mathcal{P}_p \backslash X_p \wedge RCS_q \wedge SN_q < SN_p \ \longrightarrow\ (RD_q^p, X_p) := (1, X_p \cup \{q\})\,,\ p_3)\,,$
$(p_3,\ q \in \mathcal{P}_p \backslash X_p \wedge [\neg RCS_q \vee SN_p < SN_q] \ \longrightarrow\ (OR_p^q, X_p) := (0, X_p \cup \{q\})\,,\ p_3)\,,$
$(p_3,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_4)\,,$
$(p_4,\ q \in \mathcal{P}_p \backslash X_p \wedge \neg OR_p^q \ \longrightarrow\ X_p := X_p \cup \{q\}\,,\ p_4)\,,$
$(p_4,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_5)\,,$
$(p_5,\ rcs_p \ \longrightarrow\ [\,],\ p_5)\,,$
$(p_5,\ \neg rcs_p \ \longrightarrow\ RCS_p := 0\,,\ p_6)\,,$
$(p_6,\ q \in \mathcal{P}_p \backslash X_p \wedge RD_p^q \ \longrightarrow\ (RD_p^q, OR_q^p, X_p) := (0, 0, X_p \cup \{q\})\,,\ p_6)\,,$
$(p_6,\ q \in \mathcal{P}_p \backslash X_p \wedge \neg RD_p^q \ \longrightarrow\ X_p := X_p \cup \{q\}\,,\ p_6)\,,$
$(p_6,\ X_p = \mathcal{P}_p \ \longrightarrow\ X_p := \emptyset\,,\ p_0)\,.$

Fig. 4. Abstract code of an intermediate version of R-A algorithm

place predicate *at* p_3 is true". Below is the formal definition of the latter and other similar predicates.

$$\begin{aligned}
at^q\, p_0 &=_{def} ((at\ p_6 \wedge q \in X_p) \vee at\ p_0)\,,\\
at^q\, p_2 &=_{def} (at\ p_2 \wedge q \in \mathcal{P}_p \backslash X_p)\,,\\
at^q\, p_3 &=_{def} ((at\ p_2 \wedge q \in X_p) \vee (at\ p_3 \wedge q \in \mathcal{P}_p \backslash X_p))\,,\\
at^q\, p_4 &=_{def} ((at\ p_3 \wedge q \in X_p) \vee (at\ p_4 \wedge q \in \mathcal{P}_p \backslash X_p))\,,\\
at^q\, p_5 &=_{def} ((at\ p_4 \wedge q \in X_p) \vee at\ p_5)\,,\\
at^q\, p_6 &=_{def} (at\ p_6 \wedge q \in \mathcal{P}_p \backslash X_p)\,.
\end{aligned}$$

The invariant I is the conjunction, for all distinct p and q, of the assertions

$$\begin{aligned}
1_p &: [X_p \subset \mathcal{P}_p \wedge (at\ p_{05} \Rightarrow X_p = \emptyset)]\,,\\
2_p &: [at\ p_{06} \equiv \neg RCS_p]\,,\\
3_{pq} &: [SN_p \neq SN_q \wedge SN_p < time \wedge ((at^q\, p_5 \wedge RCS_q) \Rightarrow SN_p < SN_q)]\,,\\
4_{pq} &: [(RD_q^p \Rightarrow OR_p^q) \wedge (\neg at^q\, p_3 \equiv (OR_p^q \Rightarrow RD_q^p))]\,,\\
5_{pq} &: [(at^q\, p_6 \wedge at^p\, q_4) \vee (RD_p^q \equiv (at^p\, q_4 \wedge RCS_p \wedge SN_p < SN_q))]\,.
\end{aligned} \tag{2}$$

Acceptable initial conditions are, for all distinct stations p and q,

$$at\ p_0 \wedge X_p = \emptyset \wedge \neg RCS_p \wedge \neg OR_p^q \wedge \neg OR_q^p \wedge \neg RD_p^q \wedge \neg RD_q^p \wedge SN_p \neq SN_q\,.$$

3.2 What can be obtained in an automatic way ?

Our task is to use CAVEAT in order to determine whether I really is an invariant of system $\mathcal{S}$. This system is typically "nearly propositional". Most of the variables are Boolean; SN_p and X_p are not, but $SN_p < SN_q$ and $q \in X_p$ are. Another worrying point is the parameter N (number of nodes in the network). Clearly enough, there exists a formula $I(p, q)$ — in fact, the conjunction of formulas (2) — such that the invariant really is

$$I \ =_{def}\ \forall p \forall q \neq p\, I(p, q)\,.$$

Due to symmetry, we can now *fix* two specific distinct stations p and q and decide that only the transitions explicitly written in Figure 4 ought to be checked against invariant I. Now, we have 14 transitions to consider instead of $O(N^2)$, but the size of the invariant is still $O(N^2)$. We cannot similarly reduce the triple $\{I\}\tau\{I\}$ to the triple $\{I(p,q)\}\tau\{I(p,q)\}$. However, we can reduce the triple $\{I\}\tau\{I\}$ to the family of $N*(N-1)$ triples $\{I\}\tau\{I(p',q')\}$. We can further observe that, what matters about p', q' are whether they belong to $\{p,q\}$ or not. Let us now assume that p,q,r,s are four distinct stations (and therefore, that $N \geq 4$). We can reduce the aforementioned family of triples to only seven triples,[5] listed below:

1. $\{I\}\,\tau\,\{I(p,q)\}$,
2. $\{I\}\,\tau\,\{I(q,p)\}$,
3. $\{I\}\,\tau\,\{I(p,s)\}$,
4. $\{I\}\,\tau\,\{I(r,q)\}$,
5. $\{I\}\,\tau\,\{I(s,p)\}$,
6. $\{I\}\,\tau\,\{I(q,r)\}$,
7. $\{I\}\,\tau\,\{I(r,s)\}$.

Triple 3, for instance, serves as a pattern for $N-2$ triples of the family since s stands for any node distinct from p and q.[6]

A similar reduction can be operated on the precondition. For instance, triple 1 can be replaced by

$$\{I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s)\}\ \tau\ \{I(p,q)\}$$

Such a triple, when fully developed, is a finite piece of text. The corresponding formula Φ is truly propositional, provided that predicates like $SN_p < SN_q$ and $q \in X_p$ are considered as atoms (we call *pseudo-atoms* these predicates; true atoms are location predicates and Boolean variables). The connection method will work, but only connections involving atoms will be detected with certainty (they contain the same atom with both polarities). Connections involving pseudo-atoms can remain undetected. For instance,

$$\{\mathbf{T} : SN_p < SN_q,\ \mathbf{F} : SN_p < SN_q\}$$

will be detected, but

$$\{\mathbf{T} : SN_p < SN_q,\ \mathbf{T} : SN_p = SN_q\}$$

will not. The simple default strategy followed by CAVEAT is to suppose that when a path can not be closed using atoms only, pseudo-atoms form a connection. Such paths are collected, and the suspected connections are put into a table to be validated by the user. Even if, say, 1000 paths contain connections involving pseudo-atoms, it is possible that only a dozen distinct connections exist. So CAVEAT should sort the paths according to suspected connections, in order to minimize the work performed by the user.

---

[5] Only four triples are needed if $N = 3$, and two triples if $N = 2$.

[6] Taking symmetry into account may allow to reduce the number of assertions and the number of transitions. The favourable case (as for this version of Ricart and Agrawala's algorithm) occurs when these numbers become true constants (independent from the size N of the network, or from any other parameter). An example of the unfavourable case is reported in [14].

3.3 What is obtained using CAVEAT ?

The main data file for CAVEAT contains the program to be verified. The declarations are rather standard and omitted here. The languages for transitions and assertions are slightly adapted from those used in Figure 4 and Formula (2).

Two differences exist between the real code and the abstract code in Figure 4. First, [ ] becomes `skip`, since it does not interfere with the invariant. Second, the set X_p is implemented as a Boolean array `XP`, with `XP[q]` meaning $q \in X_p$ (we suppose that `XP[p]` is true, although that does not really matter). The constant `XPempty` is such that `XPempty[q]` holds only if `q=p`; the variable `XPCard` records the number of true elements in the array `XP`. The transformation induced in the real code is straightforward.

The main data file also contains the invariant to be verified.

CAVEAT generates and explores the VAG for each of the 14 transitions; if the invariant to be checked is $I(p,q) \wedge I(q,p)$, this takes ten minutes (SUN Sparc 10) since, in spite of the simplifications introduced above, the path list remains long.[7] However, most of the paths are closed by the system, and the set of "suspected connections" submitted to the user is short: 11 small sets, all of which being inconsistent. Here they are, in abstract notation:

1. $\{\mathbf{F} : SN_q = SN_p,\ \mathbf{F} : SN_q < SN_p,\ \mathbf{F} : SN_p < SN_q\}$,
2. $\{\mathbf{T} : SN_p < SN_q,\ \mathbf{T} : SN_q < SN_p\}$,
3. $\{\mathbf{F} : q \in X_p,\ \mathbf{T} : |X_p| = N\}$,
4. $\{\mathbf{T} : q \in \{p\}\}$,
5. $\{\mathbf{T} : SN_q < time,\ \mathbf{T} : time = SN_q\}$,
6. $\{\mathbf{T} : |X_q| = 1,\ \mathbf{T} : p \in X_q\}$,
7. $\{\mathbf{F} : time < time + 1\}$,
8. $\{\mathbf{T} : |X_p| = 1,\ \mathbf{T} : q \in X_p\}$,
9. $\{\mathbf{T} : SN_q < time,\ \mathbf{T} : time < SN_q\}$,
10. $\{\mathbf{T} : SN_q < time,\ \mathbf{T} : SN_q = time\}$,
11. $\{\mathbf{T} : SN_q < time,\ \mathbf{F} : SN_q < time + 1\}$.

In this favourable case, CAVEAT succeeds in isolating exactly the (tiny) non-propositional part of the verification work; in order to understand the program, the user has to know that all the eleven small lists of formulas given above are inconsistent.[8]

3.4 Limitations of CAVEAT

Within the restricted, but important subclass of programs CAVEAT is intended to validate, two worrying limitations have been found. First, the gap of running time between the short version of the invariant, i.e.

$$I(p,q) \wedge I(q,p)$$

and the full version, i.e.

[7] Nearly twenty hours are needed for the full version of the invariant, i.e.

$$I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s).$$

[8] Observe that, although the invariant is symmetric w.r.t. p and q, the code is not, and neither is the connection set.

$$I(p,q) \wedge I(q,p) \wedge I(p,s) \wedge I(r,q) \wedge I(s,p) \wedge I(q,r) \wedge I(r,s),$$

is clearly not acceptable (ratio is worse than 1 to 100), especially since the last five assertion groups of the full version are mostly trivial. This limitation prevents us for now to consider larger, more realistic systems. Techniques for decomposing invariants are currently investigated.

Second, CAVEAT is not efficient for parametric systems whose parameter is not the number of processes. An example is Stenning's "sliding window" protocol, where the parameter is the size of the window. The problem is that quantification elimination is more difficult in this case, and leads to longer propositional formulas.

3.5 A necessary extension

CAVEAT is inspired by the classical idea that the best way to validate (the safety part of) the specifications of a concurrent system is to provide an appropriate invariant. However, as many designers are already reluctant to write specifications in a formal way, they are even less likely to be willing to also provide the invariant. Indeed, although the invariant is usually not more complex than the program code, it is more complex than the specification and not obvious to derive. The conclusion is that the construction of the invariant itself should be automated as much as possible.

The point of view adopted in [14, 16] is to view the system under study, say $\mathcal{S}_n$, and its invariant I_n, as the last pair of a sequence $((\mathcal{S}_k, I_k) : k \leq n)$ of "specified systems". Small transformation steps lead quite systematically from one version to the next, and the initial system $\mathcal{S}_0$ is very abstract, so the construction of its invariant I_0 is usually easy. As can be seen in [16], the design/verification process is quite lengthy but *much more time was devoted in verifying the "candidate-invariants" (by hand) than in their actual construction*; this construction will be integrated in the next version of CAVEAT. Note however that the construction process is not always amenable to automation. The exercise considered in [14] is probably a worst case in this respect. The extension to liveness and other temporal properties may be possible, using e.g. the technique reported in [15].

4 Related work

Several successful experiments have been made in combining model checking and theorem proving. In [19], an 8.2^m-bit multiplier is verified in this way. The principle is to verify the basic component of the multiplier, i.e., the 8-bit multiplier, by model-checking. Theorem proving (in temporal logic) is used to validate the recursive way in which four N-bit multipliers are combined to form a $2N$-bit multiplier. This approach takes full benefits of the now powerful implementations of model-checking algorithms, but applies to a more restricted class of programs than ours. The reason is that many parametric systems (including Ricart and Agrawala's algorithm) cannot be decomposed into non-parametric ones.

In this paper, we avoid this decomposition problem and consider the system to be verified as a whole. Simplification is performed on the verification conditions. As a result, we do not use model-checking, but tautology-checking.

Our approach is more similar to the approach reported in [25]. The system STeP uses model-checking whenever possible, and reverts to (temporal) theorem proving when model-checking fails. STeP does not rely on our incremental approach for obtaining invariants, but attempts to synthesize invariants directly from the program code. It also integrates various simplification methods, including two decision procedures for Presburger arithmetic (the first one is efficient, the second one is complete). STeP does not appear to achieve a full separation between the automatic part and the ATP-supported part, which is one of our main objectives. Indeed, in our opinion, this separation allows to reduce the ATP-part to short and elementary formulas, for which complicated ATP techniques are not really needed.

The incremental approach that is currently integrated in CAVEAT is not the only way to transform concurrent systems, from higher-level to lower-level versions. Other approaches might be amenable to partial automation, for instance those refining atomicity with a reduction principle [2, 22], those using refinements and hierarchical design [24, 18] or phase decomposition [10, 28], and those based on property preserving abstractions [4, 12].

Symbolic model-checking and tautology-checking can be improved by using (ordered) binary decision diagrams [6]. This approach is followed in [7], and successfully applied to the verification of a simple synchronous pipeline. Besides, using Boolean automata can be more effective than using Boolean formulas, and this kind of approach is not restricted to investigating concurrent systems [17]. First experiments with (O)BDD in CAVEAT have not been encouraging, however, since we lack an effective procedure for ordering atoms and pseudo-atoms. No experiment has been made yet in the area of digital circuits.

References

1. K.R. Apt and D.C. Kozen, Limits for Automatic Program Verification, *Inform. Process. Letters* **22** (1986) 307-309.
2. R.J. Back, A Method for Refining Atomicity in Parallel Algorithms, PARLE'89, *Lect. Notes in Comput. Sci.* **366** (1989) 199-216.
3. R.J. Back and R. Kurki-Suonio, Distributed co-operation with action systems, *ACM Trans. Programming Languages Syst.* **10** (1988) 513-554.
4. S. Bensalem et al., Property Preserving Abstractions for the Verification of Concurrent Systems, to appear in *Formal Methods in System Design* (1994).
5. W. Bibel, *Deduction – Automated Logic*, Academic Press, 1993.
6. R.E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Trans. on Computers* **C-35** (1986) 677-691.
7. J.R. Burch et al., Symbolic Model Checking: 10^{20} States and Beyond, *Proc. 5th. Symp. on Logic in Computer Science* (1990) 428-439.
8. 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.

9. E.W. Dijkstra and al., On-the-Fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM* **21** (1978) 966-975.
10. T. Elrad and N. Francez, Decomposition of Distributed Programs into Communication-closed Layers, *Sci. Comput. Programming* **2** (1982) 155-173.
11. D.M. Goldschlag, Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover, *IEEE Trans. on Software Eng.* **16** (1990) 1005-1023.
12. S. Graf, Verification of a distributed Cache memory by using abstractions, *Lect. Notes in Comput. Sci.* **818** (1994) 207-219.
13. E.P. Gribomont, Synthesis of parallel programs invariants, TAPSOFT'85, *Lect. Notes in Comput. Sci.* **186** (1985) 325-338.
14. E.P. Gribomont, Stepwise refinement and concurrency : the finite-state case, *Sci. Comput. Programming* **14** (1990) 185-228.
15. E.P. Gribomont, Design, verification and documentation of concurrent systems, in *Proc. 4th. Refinement workshop*, J.M. Morris and R.C. Shaw (Eds), pp. 360-377, Springer-Verlag, 1991.
16. E.P. Gribomont, Concurrency without toil : a systematic method for parallel program design". *Sci. Comput. Programming* **21** (1993) 1-56.
17. N. Halbwachs and F. Maraninchi, On the symbolic analysis of combinational loops in circuits and synchronous programs, REACT Report, 1994.
18. B. Jonsson, Compositional Specification and Verification of Distributed System, *ACM Trans. Programming Languages Syst.* **16** (1994) 259-303.
19. R.P. Kurshan and L. Lamport, Verification of a Multiplier : 64 Bits and Beyond, CAV'93, *Lect. Notes in Comput. Sci.* **697** (1993) 166-179.
20. R.P. Kurshan and M. McMillan, A structural induction theorem for processes, *Proc. 8th ACM Symp. on Principles of Distributed Computing*, Edmonton (1989).
21. L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm, *Sci. Comput. Programming* **2** (1983) 175-206.
22. L. Lamport and F.B. Schneider, Pretending Atomicity, DEC SRC Rep. 44, May 1989.
23. R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO : A High-Performance Theorem Prover, *Jl. of Automated Reasoning* **8** (1992) 183-212.
24. N.A. Lynch and M.R. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, New-York (1987) 137-151.
25. Z. Manna et al., **STeP** : the Stanford Temporal Prover (Draft), June 1994.
26. J.S. Moore, Introduction to the OBDD algorithm for the ATP Community, *Jl. of Automated Reasoning* **12** (1994) 33-45.
27. G. Ricart and A.K. Agrawala, An optimal algorithm for mutual exclusion, *Comm. ACM* **24** (1981) 9-17 (corrigendum : *Comm. ACM* **24** (1981) 578).
28. F. Stomp and W.P. de Roever, A principle for sequential phased reasoning about distributed systems, *Formal Aspects of Computing* **6** (1994) 716-737.
29. M.Y. Vardi, P. Wolper, An Automata-Theoretic Approach To Automatic Program Verification, *Proc. Symp. on Logic in Comput. Sci.*, Cambridge (1986) 322-331.
30. P. Wolper and V. Lovinfosse, Verifying Properties of large Sets of Processes with Network Invariants, CAV'89, *Lect. Notes in Comput. Sci.* **407** (1990) 68-80.
31. L. Wallen, *Automated Deduction in Nonclassical Logics*, MIT Press, 1990.