A tool for symbolic program verification and abstraction*

Susanne Graf and Claire Loiseaux

VERIMAG, BP 53X, F-38041 Grenoble, e-mail : {graf,loiseaux}@imag.fr

Abstract: We give the description of a verification tool taking boolean programs of guarded commands as input; internal representation of programs are sets of Binary Decision Diagrams (BDD) (one for each guarded command). It allows to construct an abstract program of the same form obtained using an abstraction relation given by a boolean expression on "concrete" and "abstract" variables. The tool allows the verification of CTL formulas on programs. We illustrate its possibilities on an example.

1 Introduction

In the domain of program verification an obvious idea is to verify some abstract program instead of the complete specification (called concrete program) depending on the properties to be verified. The motivation is to make the representation of the program model smaller and this for two reasons: one is to make the verification faster; the other is that in most practical cases the model of the concrete program is too large to be verified, whereas an abstraction of it may be sufficiently small and still contain sufficient information with respect to the properties to be verified.

However, this approach rises the problem of property preservation, i.e., we have to know which properties holding on the abstract program hold also on the concrete one. The investigation of property preserving abstractions of reactive systems has been the object of intensive research during the last years. Results have been given e.g. in [Kur89,CGL91,BBLS92,GL92].

One way to define abstractions is via a behavioral equivalence, such as observational equivalence [Mil80]; in this case, an abstract program can be calculated by constructing an equivalent program which is minimal with respect to the used equivalence by using for example the algorithm of minimal model generation given in [Fer90] or [BFH90]. These algorithms calculate the largest possible partition on the domain (set of states) of the program, such that the following program is equivalent to the original program: take as domain the set of the calculated classes, and as transition relation the one relating two abstract states if and only if two elements in the corresponding classes are related. The advantage of this method is that for a large class of properties, the abstract program satisfies a property *if and only if* the concrete program satisfies it (i.e. one has strong preservation); its disadvantage is the high cost you have to pay in order to get such an abstract program.

Here, we present a tool implementing the ideas presented recently in [BBLS92,GL92] and before in [Sif83] and in some sense also in [CC77]. Instead of calculating the largest partition on the domain of the concrete program, such that the obtained abstraction is

---

* This work was partially supported by ESPRIT Basic Research Actions "SPEC" and "REACT"

equivalent, we give an arbitrary partition by defining a relation ρ between the concrete and some new *abstract domain*. In this case, the abstract program calculated in the same manner as above is no longer equivalent, but the concrete program does simulate it in the sense of [Mil71]. This means that for all the safety properties and even more (see Section 3) one has that if the abstract program satisfies a property, the concrete one also does. However, so calculated abstract program are obtained much easier, and may also be much smaller.

This gives, at least in the case we consider only finite domains, a framework of automated program verification: given a concrete program on domain (set of states) D and a relation ρ relating concrete domain D and abstract domain D_A and a set of properties to be verified, all the rest can be done automatically, that is:

- check that properties to be verified are preserved by the used abstraction,
- calculate the corresponding abstract program,
- verify the property on the calculated abstract program.

The tool we have currently implemented works on *boolean guarded command programs* which may be composed by different parallel composition operators. It can be envisaged to extend this approach also to more interesting calculi which are not necessarily defined on finite domains but which must be decidable (and they should also be "reasonably implementable"). Another possible extension is to translate a significant subset of Lotos [BB88] programs into composed guarded command programs which is almost straightforward and gives the tool a wide range of applications.

A tool based on similar ideas is suggested in [CGL91] and has been implemented; there, macros for n-bit integers and corresponding integer operations but no parallel composition have been defined. The main difference is that there only a much more restricted set of abstraction relations are allowed (each variable must be abstracted independently) for which the abstract program can be computed easily.

In the next section, we give the principles of our tool, define parallel operators and show how abstract programs are computed from a relation between states. In Section 3, we recall the results on property preservation given in [BBLS92,GL92]. In Section 4, we give the syntax of the programming language and the formula language accepted by the tool. In Section 5 we treat an example and give some results concerning the performances of the tool.

2 The tool

The tool takes as input a description of a concrete program in the form of a composition of boolean guarded command programs using operations of parallel composition and abstraction as described in Section 4. An abstraction is given by means of a boolean expression on abstract and concrete program variables and represents a relation between "concrete" states (those of the program) and "abstract" states. The tool computes an abstract program such that the concrete program simulates the abstract one in the sense of Milner [Mil71] for the given abstraction relation. Notice that parallel composition and abstraction can be applied in any order.

On so obtained abstract programs, the tool verifies properties given by formulas of CTL [CES83]. If the formula is given using predicates of a concrete program, the tool computes their interpretation in terms of abstract variables.

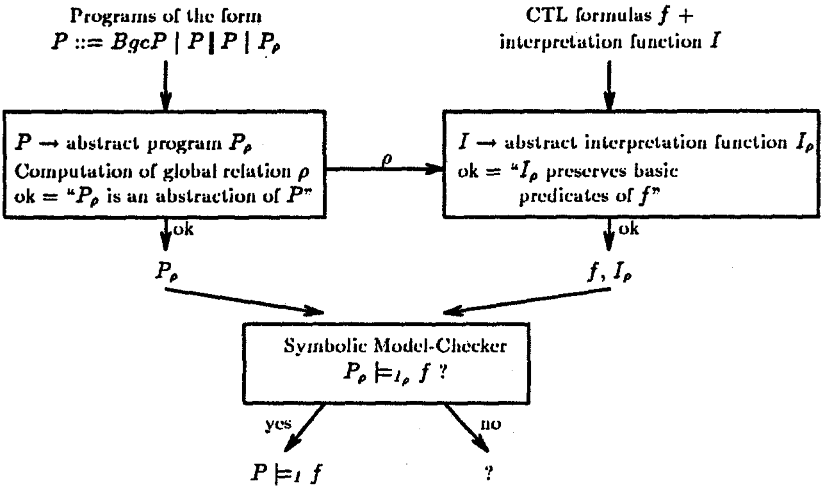
Programs of the form
$P ::= BgcP \mid P \| P \mid P_\rho$
CTL formulas f +
interpretation function I
$P \rightarrow$ abstract program P_ρ
Computation of global relation ρ
ok = "P_ρ is an abstraction of P"
ρ
$I \rightarrow$ abstract interpretation function I_ρ
ok = "I_ρ preserves basic predicates of f"
ok
ok
P_ρ
f, I_ρ
Symbolic Model-Checker
$P_\rho \models_{I_\rho} f$?
yes
no
$P \models_I f$
?

Fig. 1. Principle of the tool

Furthermore, for any property to be verified on an abstract program, one must verify preservation by the given abstraction relation which depends on two criteria:

- the subset of CTL used,
- the atomic predicates occurring in the formula.

The first point is a simple syntactic criterion and its verification is left to the user, whereas the preservation of predicates is verified by the tool (the exact preservation criteria are given in Section 3).

2.1 Representation choices

A guarded command program defines a [set of] transition relation[s] on the set of valuations of its variables (which we call "domain"); any CTL formula defines a set of states (elements in the domain), the states in which it holds. To represent relations, we need a formalism allowing to do all necessary operations, such as "compare", "successors", ... with a reasonable amount of time and space. It is now well established that Binary Decision Diagrams (BDDs) [Bry86] deserve this objective for sets (of states, transitions, ...) represented by boolean expressions.

We never represent the global transition relation, but one for each command. The main reason is that the BDD encoding the global relation is likely to be much larger than the set of BDDs encoding each command.

In order to represent transition relations, i.e. sets of edges represented by pairs of states, a double set of the program variables is necessary, one (called this-state variables) coding the source and the other (called next-state variables) coding the target state of each transition.

A well-known characteristics of BDDs is that their size is very sensitive to the variable ordering. Two heuristics are implemented in the tool. The first one is that in all cases each next-state variable follows directly its corresponding this-state variable as it has been suggested e.g. in [EFT91,BD92]; the other is that global variables (shared by several processes) come before local variables. As good heuristics are in general strongly dependent on the underlying transition system, we give the user the possibility to define his own variable ordering.

The user can also define an "invariant" restricting the domain to the set of states satisfying this invariant; this allows to obtain smaller BDDs representing the transition relations. by using the "restrict" operator on BDDs which has been implemented in the BDD package we use [Rat92]. However, the use of this facility is under the plain responsibility of the user, as the computed transition relation is an abstraction of the concrete one only if all reachable states are contained in the invariant.

2.2 Parallel composition

We suppose a universal set of program variables $\mathcal{V}$. Any transition relation R is defined on the set of states which is the domain $Dom(X)$ of some subset X of $\mathcal{V}$. Where, e.g., for $X = \{x, y\}$, $Dom(X) = Dom(x) \times Dom(y)$.

As already mentioned, binary relations on $Dom(X)$ can be represented symbolically by predicates of the form $R(X, X')$ (encoded by BDDs in our tool) where X' is a "copy" of X, i.e. $Dom(X) = Dom(X')$, X encoding the source state and X' the target state of any transition in R; e.g., if $Dom(x) = \mathcal{N}$, then "$x' = x + 1$" represents the transition relation relating any $n \in \mathcal{N}$ with $n + 1$. This approach is used e.g. in [Lam91,Pnu86]. In the same way a relation ρ from $Dom(X)$ to $Dom(X_A)$ is represented by a binary predicate of the form $\rho(X, X_A)$.

In this setting, set operations are expressed by logical connectives. E.g., the fact that a relation R is included in R' is expressed by $R \Rightarrow R'$ and $R \wedge R'$ represents the intersection of R and R' if they are defined on the same set of variables.

We consider that a *program* is a family of transition relations represented by sets of binary predicates on the same set of variables $S = \{\ R_i(X, X')\ \}_{i \in I}$ where the elements of I are labels used for synchronization purposes in parallel composition in Section 2.2.

Definition 1. Let be X, Y sets of program variables. Let be $S_1 = \{R_{1i}(X, X')\}_{i \in I}$ and $S_2 = \{R_{2j}(Y, Y')\}_{j \in J}$ be programs on $Dom(X)$ respectively $Dom(Y)$. Let $A \subseteq I \times J$ be a synchronization set and $A_I = \{i | \exists j.(i, j) \in A\}$ and $A_J = \{j | \exists i.(i, j) \in A\}$ the projections of A on I respectively J. Then, parallel compositions of S_1 and S_2 are programs on $X \cup Y$ defined as follows:

- Synchronous Composition : $S_1 \otimes_A S_2 = \{R_{1i}(X, X') \wedge R_{2j}(Y, Y')\}_{(i,j) \in A}$
- Asynchronous Composition : $S_1 \| S_2 =$
  $\{R_{1i}(X, X') \wedge \forall y \in (Y - X).(y' = y)\}_{i \in I} \cup \{R_{2j}(Y, Y') \wedge \forall x \in (X - Y).(x' = x)\}_{j \in J}$
- Mixed Composition : $S_1 [A] S_2 =$
  $\{R_{1i}(X, X') \wedge R_{2j}(Y, Y')\}_{(i,j) \in A} \cup$
  $\{R_{1i}(X, X') \wedge \forall y \in (Y - X).(y' = y)\}_{i \notin A_I} \cup \{R_{2j}(Y, Y') \wedge \forall x \in (X - Y).(x' = x)\}_{j \notin A_J}$

Comments :

- In synchronous composition, each transition of the composed program results from the synchronous execution of two transitions of the transition relations defined by some R_{1i} and some R_{2j} for $(i, j) \in A$. If $A = I \times J$, this operator corresponds exactly to conjunction applied on programs described by TLA formulas [Lam91]; it is also very similar to program composition of S/R models [KK86]. If the domains of the component processes are independent, it is exactly the one of SCCS [Mil83].

- In the asynchronous composition, in each step one of the programs executes one of the currently enabled transitions and the other idles. This operator results in the complete "interleaving" of the component processes if they are defined on independent domains; if not, the execution of a transition of one of the processes may change the enabling conditions of the other one. This operator is in fact the union operator of Unity [CM88].
- Finally, in the mixed composition operator, some of the transitions must be executed synchronously, whereas the others are executed asynchronously. This operator is not exactly the one in CSP [Hoa84] or Lotos [BB88], where all processes have distinct variable sets and communicate by exchanging values; however, it is equivalent to them.

2.3 Computation of abstract programs

Let Y be a set of abstract variables and $Dom(Y)$ its associated domain. Then an abstraction relation is given by a predicate of the form $\rho(X, Y)$ on concrete and abstract variables (this can be seen as a kind of abstract interpretation).

Definition 2. The set of transition relations representing the abstract program calculated from $S = \{R_i(X, X')\}_{i \in I}$ and ρ is

$$S_\rho = \{R_{i\rho}(Y, Y') \mid i \in I\} = \{\exists X \, \exists X' \, . \, \rho(X, Y) \wedge R_i(X, X') \wedge \rho(X', Y')\}_{i \in I}.$$

If all the R_i as well as ρ are represented by BDDs, the BDDs representing the $R_{i\rho}$ are obtained by applying usual operators on BDDs. Notice that in case of finite domains, the variables X and X' can always be eliminated.

In Section 3, we show that the so calculated abstract program is in fact well chosen. For a large class of properties, we have property preservation from the abstract to the concrete program.

2.4 Verification of CTL formulas

For the verification of properties, given by CTL formulas, the global transition relation is needed. But as formulas are in general verified on the abstract program this is acceptable. The implemented model checking algorithm is a symbolic one using the fixed point definitions of the operators of CTL, e.g.,

$$\| EG(f) \| = \bigwedge P_i$$

where $P_0 = \| f \|$, the set of states satisfying f (represented by a BDD on the set of program variables), and $P_{i+1} = P_i \wedge pre[R_G](P_i)$ (where $pre[R_G]$ is the inverse image function of the global transition relation R_G). Thus, the characteristic set of $EG(f)$ is computed by first computing the characteristic set of f and then adding successively the set of states having successors in the so far computed set. The fixed point is reached when $P_{i+1} = P_i$ and as the domain is finite the fixed point is always reached. All these computations can be done symbolically for any program of the form $S = \{R_i(X, X')\}_{i \in I}$ and for any predicate $p(X)$ (encoded by BDDs) we have

$$pre[R_G](p)(X) = \bigvee_i \exists X' . R_i(X, X') \wedge p(X')$$

This is a predicate with free variables in X (all the quantified variables X' can be eliminated), and all the used operators are available in most BDD-packages.
Notice that despite the fact that we need the inverse image function of the global transition relation, we do not need a unique BDD representing it, but we can still work with the before calculated *set* of BDDs representing the global transition relation.

3 Preservation Results

In this section we give a brief overview on the preservation results, presented in [BBLS92, GL92], guaranteeing that properties verified by the tool on an abstract program hold also on the corresponding global concrete program. Notice also that if a property does *not* hold on the abstract program, nothing can be said in general on its validity on the concrete program, and one should try with a different abstraction e.g., by refining the already given one. We need three results, concerning

- the subset of the CTL preserved by abstraction,
- the set of atomic predicates preserved by an abstraction relation ρ,
- compositionality of abstraction with respect to parallel composition.

Definition 3. [BBLS92] Let $R(X, X')$ and $R_A(X_A, X'_A)$ be transition relations (the global transition relations associated with a concrete and an abstract program), and $\rho(X, X_A)$ an abstraction relation. Then,

R ρ-simulates R_A iff $R^{-1}\rho \subseteq \rho R_A^{-1}$

Notice that "there exists ρ such that ρ-simulates" defines the standard simulation preorder [Mil71]. For a given R and ρ we are interested in a abstraction relation R_A that is as close as possible to the given R, i.e. that contains no more transitions than necessary. The least R_A does not always exist, but we have:

Proposition 4. *[GL92,BBG*93] (S_ρ is a reasonable abstraction)*
For $S = \{R(X, X')\}_{i \in I}$ and $\rho(X, X_A)$ total on Dom(X), the program S_ρ as defined in Definition 2 one has that S ρ-simulates S_ρ.

If furthermore $\rho \circ \rho^{-1} \circ \rho = \rho$ holds (which means that ρ is a function from a partition on Dom(X) to a partition on Dom(X_A) and is true in most practical cases), then S_ρ is the smallest ρ-abstraction of S modulo bisimulation in the sense of [Mil80]; if ρ is a function, R_ρ is the least ρ-abstraction.

Theorem 5. *[BBLS92] Let $R(X, X')$ and $R_A(X_A, X'_A)$ be transition relations and $\rho(X, X_A)$ an abstraction relation total on Dom(X), $I : \mathcal{P} \rightarrow 2^{Dom(X)}$ an interpretation function of atomic predicates on Dom(X) such that R ρ-simulates R_A.*

ACTL is the subset of CTL [CES83] containing the formulas without negation and using only universal quantification on paths. For any $f \in ACTL$,

$$R_A \models_{pre[\rho^{-1}] \circ I} f \quad \textit{implies} \quad R \models_{pre[\rho] \circ pre[\rho^{-1}] \circ I} f$$

If furthermore, $pre[\rho] \circ pre[\rho^{-1}] \circ I\ (p) = I(p)$ (ρ preserves *I) for any predicate symbol p occurring in f, we have even $R \models_I f$.*

This means: "if f holds on the abstract program, taking as interpretation function $pre[\rho^{-1}] \circ I$, associating with any atomic predicate p the image by ρ of its interpretation by

I on the concrete domain, then on the concrete program f holds taking as interpretation function $pre[\rho] \circ pre[\rho^{-1}] \circ I$, obtained from I by translating it forth and back by ρ". It yields for each predicate an equal or larger interpretation than I. Thus, in order to obtain the initially required result $R \models_I f$, we need to make sure that each basic predicate in f translated forth and back by ρ remains unchanged.

This condition is verified by our tool where atomic predicates are the (boolean) program variables with the obvious interpretation function.

The last result we need concerns the compositionality of abstraction with respect to parallel composition. The following theorem gives for all the parallel composition operators conditions under which the composition of abstract programs computed by our tool is an abstraction of the composed program (for a complete result see [GL92]).

Notations: In the following we consider a set of variables $X = X_1 \cup X_2$, two transition systems $S_i = \{R_{ij}(X_i, X_i')\}_{j \in I_i}$, $i \in \{1,2\}$ and $X_A = X_{1A} \cup X_{2A}$ a set of abstract variables. We denote also $X_c = X_1 \cap X_2$, the set of common variables, $X_{il} = X_i - X_c$ the local variables of S_i and analogously $X_{cA} = X_{1A} \cap X_{2A}$, the set of common abstract variables and $X_{ilA} = X_{iA} - X_{cA}$ the local abstract variables of S_i.

Theorem 6. *[GL92] Let $\rho_1(X_1, X_{1A})$ and $\rho_2(X_2, X_{2A})$ be abstraction relations such that $\rho_1 \wedge \rho_2$ is total on $Dom(X)$. We have,*

$(\otimes) R_1 \otimes_A R_2$ $\rho_1 \wedge \rho_2$-simulates $R_{1\rho_1} \otimes_A R_{2\rho_2}$,

Suppose that $\rho_i(X_i, X_{iA})$, $i = \{1,2\}$ can be decomposed as $\rho_i((X_{il}, X_{ic}), (X_{ilA}, X_{icA})) = \rho_{il}(X_{il}, X_{ilA}) \wedge \rho_{ic}(X_i, X_{icA})$ (values of abstract local variables depend only on values of concrete local variables, whereas the values of abstract common variables may depend on the values of all concrete variables). Then,

$(\|)\ R_1 \| R_2$ $\rho_1 \wedge \rho_2$-simulates $R_{1\rho_1} \| R_{2\rho_2}$
$([])R_1 [A] R_2$ $\rho_1 \wedge \rho_2$-simulates $R_{1\rho_1} [A] R_{2\rho_2}$

This gives for each parallel operator a necessary condition in order R_ρ to be used for the verification of properties of R; none of these conditions is very strong. It is even not necessary that the two abstraction relations coincide exactly on the common domain, they have just to coincide sufficiently in order $\rho_1 \wedge \rho_2$ to be total on D. Notice that in the case where nothing about the transition relations R_i is known these conditions are also necessary.

The theorem generalizes well-known results concerning compositionality of abstraction in process algebras, and this in two ways. One is that in the here presented results simulation is parameterized by an abstraction relations which gives a more precise compositionality result. The other is that also processes with common variables are taken into account.

4 Syntax of the input language of the tool

A program, input of our tool, is an expression of the form

PROG ::= gcp | PROG ||| PROG | PROG * PROG | PROG |[label-list]| PROG |
PROG[abstract-exp] | PROG[label-list,label-list]

where "gcp" are filenames containing guarded command programs, ||| is the Asynchronous, * the synchronous and |[]| the mixed parallel operators defined in Section

2.2, except that the parameter "label-list" of |[]| is a list of labels instead of a list of pairs of labels meaning that programs synchronize on all the "labels" (see below) in this list which therefore must occur in both programs. PROG[label-list1,label-list2] defines a renaming operator. PROG[abstract-exp] defines the abstraction operator.

Each guarded command program represents a process, where all variables are global. We call a variable local to some process if it is not *used* by any other process. Certainly, this is a quite primitive "programming language", but it is sufficient to show the possibilities of the tool. We intend to add a mechanism similar to the abstract data type definition in Lotos with a possibility of automatic implementation of any variable of any finite type as a set of boolean variables with the necessary operations on it.

Each guarded command program consists of a header and a list of guarded commands. The header contains variables and commands numbers and eventually a variable ordering and an invariant restricting the domain. Each guarded command is of the form:

> '[' label ']' expression-X → expression-Xx
> where
> "label" is used to identify the command in parallel composition and renaming,
> "expression-X" a guard, which is a predicate defined on this-state variables and
> "expression-Xx" defines the transition relation by an expression on this-state and next-state variables.

The this-state variables are denoted by upper-case first letter identifiers ("ident-X") whereas the corresponding next-state variables ("ident-x") are denoted by the same identifier with the corresponding lower-case first letter. Thus,

> "expression-X" is an expression constructed with usual boolean operators: '=' (equivalent), '=>' (implies), '+' (or), '.' (and), 'NOT', on this-state variables and
> "expression-Xx" is an expression as above on both current and next-state variables and using also the macro-notations 'ON', 'OFF', 'ANY' applied to lists of this-state variables, meaning that the corresponding set of next-state variables is respectively uniformly *true*, *false* or unspecified.

Notice that due to the fact that we use boolean expressions to define transition relations, any variable not mentioned in "expression-Xx" may take any value in the next state and in order to specify that some variable keeps its value, we have to say this explicitly. This is often quite tedious; therefore, we preferred to allow as default that unmentioned variables are unchanged and have added the operator ANY to define unspecified values. The operators "ON" and "OFF" are useful to (re)set a set of variables.

> "abstract-exp", the parameter of the abstraction operator is either an expression using the same operators as "expression-X" and the operator 'ABS' on the this-state variables of the concrete program and another set of this-state variables (defining the abstract domain).

ABS is applied to a list of the concrete variables, meaning that these variables are not taken into account at all in the abstract transition relation (application of existential quantification), whereas for all variables X which do not occur at all in the expression, a new variable X_a and the constraint "$X_a = X$" is added. We have introduced this convention which is the opposite of the boolean expression way of defining abstraction as it is often very tedious to introduce a new variable for all the variables conserving their meaning and specifying them as equivalent to the old ones; we preferred to leave this task to the tool.

Syntax of the Logic The formulas that can be evaluated by the tool on a given program are CTL formulas on atomic predicates built on the program variables. In the present case, where only boolean variables are used, the atomic predicates are just the variables themselves. In order to take into account the labels in the formula without changing the logic, we have, similar as in XESAR [RRSV87], defined atomic predicates of the form

ident-X | 'sink' | 'enable'(label)

where "sink" represents the set of deadlock states, "enable(label)" represents the set of states in which one of the guarded commands labeled by "label" is enabled (this is the disjunction of the corresponding guards). Notice that formulas can only use this-state variables as otherwise we need also a "next-next" relation.

Using this definition of atomic predicates, the set of formulas is obtained by application of the boolean operators used in the program syntax CTL temporal operators : the unary next state operators EX, AX, the unary operators AG (always), EF (possibly), AF (inevitably), EG and the binary operators AU and EU (until on all respectively on some path). Furthermore, we defined macros allowing to give the formulas also in "μ-calculus" style [Koz83] or in LTAC style [RRSV87].

5 Overtaking Example

5.1 Description of the problem

The example we develop here is that of a simple overtaking protocol which has been described in Lotos in [EFJ90] and verified using the verification tool Caesar [GS90]. We have "translated" the Lotos specification into a parallel guarded command program, input of our tool and have verified properties on it. Most of these formulas could be verified by using some abstractions. As we have already mentioned, the translation from the Lotos program into a parallel guarded command program could be done automatically.

The protocol coordinates overtaking of vehicles on a road. In a vehicle queue, each vehicle can communicate with its immediate preceding and succeeding vehicle via two communication mediums. A driver who intends to overtake (client vehicle) initiates a protocol entity in his own vehicle which in turns initiates a negotiation with the vehicle in front (server vehicle). The environment decides whether overtaking is possible. If yes, the server sends the client a positive answer else the server waits for the next overtaking request. In case of a positive answer, actual overtaking takes place. It consists in exchanging, via a perfect communication medium this time, the information about their succeeding respectively preceding vehicles. Once a client has initiated an overtaking negotiation, it will keep on trying until it succeeds in overtaking, i.e. an overtaking cannot be aborted.

5.2 Program describing the system

The program modeling the scenario is a parallel composition of guarded command programs describing a set of vehicles CAR_i, communication mediums M and OT for the negotiation and overtake phases, an environment E and a set of timers T_i associated with CAR_i.

Each process CAR_i is identified by its name (Id_i), its position (Pos_i), its M-address (M_o_i) and that of its preceding vehicle (M_p_i) finally its OT-address (Ot_o_i) and that of the preceding vehicle (Ot_p_i).

Communication through a medium is modeled using some global data variable X: in a first step the emitter and the medium synchronize, the emitter sets the variable X and the medium becomes busy. In a second step the medium synchronizes with the receiver, the receiver reads X and the medium becomes free again. Notice that the medium M, used during negotiations, may lose messages whereas the medium OT used for exchanging data during actual overtaking phase is assumed to be perfect.

The environment E is modeled by a simple process switching spontaneously between states *ok* in which overtaking is allowed and *not_ok* in which it is not.

The timers T_i have two states *active* and *inactive*. They can be started in *inactive*; in *active* they can be stopped or timeout may occur. They are used by the client part of each vehicle which has to repeat its request if the authorization to overtake from the server has not been received within a certain delay (here we model just a non deterministic choice between "received in time" and "timeout occurs"). So, the global system is described by the expression

$$(\|_{i\in\{1,2,3,4\}} (CAR_i\,[A_T]\,T_i))\,[A]\,(M \,\|\, OT \,\|\, E)$$

where $A_T = \{T\_B, T\_S, T\_O\}$ and $A = \{C\_M, M\_C, C\_OT, OT\_C, E\_C_i,\ i \in \{1,2,3,4\}\}$ C_M and M_C are used for the synchronizations between the vehicles and M, C_OT and OT_C for the synchronizations between the vehicles and OT and finally E_C_i for the synchronizations between the vehicles and E.

Shared variables As we have already mentioned, variables are called global if they are shared by at least two processes and local if they appear in only one process. Here we give the list of global variables, their meaning and the processes they appear in (these variable declarations are not necessary in the program). Notice that all the variables declared as *m_address*, *ot_address*, *position* or *identity* are implemented by a pair of boolean variables (4 different values), but represented here for simplicity by a single variable.

- Variables shared between all vehicles and medium M:
  M_send, *M_dest* : *m_address* (the sender and receiver of the communication)
  M_ot_req, *M_ot_ok* : transmitted bits
- Variables shared between all vehicles and medium OT:
  OT_send, *OT_dest* : ot_address (the sender and receiver of the communication)
  OT_id : identity (transmitted message)
- Variables shared between vehicles CAR_i and E:
  E_ok : bit (the environment is in a ok mode)
  E_ind_i : bit (the vehicle asks allowance for its following vehicle to overtake)

In the following we give only the guarded command program CAR_i representing the behaviour of a vehicle as other programs are quite simple.

5.3 Guarded command program for a vehicle

Each vehicle CAR_i has the knowledge of both addresses of its preceding vehicle (M_p_i, Ot_p_i). It has also auxiliary variables to store informations temporarily.

The natural description would consist in defining each process CAR_i such that it describes a vehicle that keeps its identity and changes its position while overtaking. We have chosen an equivalent solution where vehicles have a fixed position and exchange identity. This is very useful as it allows to use much stronger abstractions in order to verify the service properties (which refer to positions but not to identities).

Local variables

- Identification variables:
  Id_i : identity, M_p_i, M_o_i : m_address of preceding, own vehicle,
  Pos_i : position, Ot_p_i, Ot_o_i : ot_address of preceding and own vehicle,
- Auxiliary variables to store data while overtaking and during negotiation:
  Aux_id_i : position, Aux_ot_i : ot_address, Aux_m_i : m_address.
- Control variables (boolean) :
  $Init_i$: initial state, $C1_i, .. , C6_i$: client states and $S1_i, .. , S5_i$: server states.

We define also an invariant that expresses that in any state at most one of the control variables is true.

Program of CAR_i

```
------------------------------ INIT PART ---------------------------------
% becomes server or restarts server
[M_C] (Init_i + S1_i + S2_i + S3_i + S4_i) . (M_dest = M_o_i) . M_ot_req ->
      OFF(M_ot_req,Init_i) . ON(S1_i) . (aux_m_i = M_snd) ;

[  ] Init_i ->  ON(C1_i) . OFF(Init_i); % spontaneously becomes client

------------------------------ CLIENT PART -------------------------------
% ignore ot_req of the following vehicle and idle
[M_C] (C1_i + C2_i + C3_i + C4_i + C5_i + C6_i + S5_i) . (M_dest = M_o_i) .
      M_ot_req -> ANY(M_dest,M_ot_req);

% ignore old ot_ok
[M_C] C1_i . M_ot_ok . (M_dest = M_o_i) -> OFF(M_ot_ok) ;

% send ot_req (ask for allowance to overtake via the Medium)
[C_M] C1_i -> (m_snd = M_o_i). (m_dest = M_p_i) . ON(M_ot_req,C2_i) . OFF(C1_i);

[T_B]  C2_i -> ON(C3_i) . OFF(C2_i); % start timer

[T_O]  C3_i -> ON(C1_i) . OFF(C3_i); % timeout: waited too long for ot_ok

% receive ot_ok (allowance to overtake via the Medium)
[M_C] C3_i . (M_dest = M_o_i) . M_ot_ok -> ON(C4_i) . OFF(M_ot_ok,C3_i);

[T_S] C4_i ->  ON(C5_i) . OFF(C4_i); % The timer is stopped

% Overtake: sends its parameters through OT_Medium to preceding vehicle
[C_OT] C5_i -> (ot_snd = OT_o_i) . (ot_dest = OT_p_i) . (ot_id = Id_i) .
       ON(C6_i) . OFF(C5_i);

% Overtake: receives parameters through OT_Medium from preceding vehicle
[OT_C] C6_i . (OT_dest = OT_o_i) -> ON(Init_i) . OFF(C6_i) .
       (id_i = OT_id) . ANY(OT_dest,OT_id);

------------------------------ SERVER PART -------------------------------
% sets overtaking indication in the environment and waits for authorisation.
[C_E] S1_i -> ON(E_ind_i,S2_i) . OFF(S1_i);

% Answer: if ok, goes on else returns in the initial state.
[E_C_i] S2_i  -> (S3_i = E_ok) .(Init_i = ~E_ok). OFF(S2_i);

% sends confirmation to the client
[C_M] S3_i -> (m_dest = Aux_m_i) . (m_snd = M_o_i) . ON(M_ot_ok,S4_i) .
      OFF(S3_i) . ANY(Aux_m_i);

% receives data from the client and stores them in its auxiliary variables.
[OT_C] S4_i . (OT_dest = OT_o_i) ->  ON(S5_i) . OFF(S4_i) .
       ANY(OT_id,OT_dest,OT_snd) . (aux_ot_i = OT_snd) . (aux_id_i = OT_id);

% sends its parameter to the client and sets its variables
[C_OT] S5_i -> ON(Init_i) . OFF(S5_i) . (ot_snd = OT_o_i) . (ot_dest = Aux_ot_i).
```

```
(ot_id = Id_i) . (id_i = Aux_id_i) . ANY(Aux_ot_i,Aux_id_i);
```

5.4 Properties

We verify the following properties on the overtaking protocol.

1. **Deadlock freedom** can be expressed by the following formula:
   $$init \Rightarrow AG(\text{NOT } deadlock)$$
   where *deadlock* is different from *sink* defined as the atomic predicate $\bigwedge_i$ NOT c_i where the c_i are all the guards of the global guarded command program obtained after executing all the parallel compositions and abstractions. *deadlock* must also contain all the states which can only do infinitely only some "useless" transitions (such as in the example the environment goes spontaneously from the state *ok* into *notok*, from *notok* into *ok*,...), that means that in a state not satisfying *deadlock* after a finite number of steps some "useful" transition must be possible. So NOT *deadlock* can be defined as $\bigvee_i EF(c_i)$ where the c_i are the enabling conditions of the set of "useful" transitions.
   init is a predicate specifying allowed initial states which has to precise the relative positions of all the cars and the corresponding M and OT-addresses in the different cars. It needs not to specify that all cars are in control state $Init_i$ but the set of specified states must be reachable. As deadlock freedom is no ACTL formula, no abstraction can be applied before its verification.
2. **Safety properties:** "If the vehicle in position i is engaged in overtaking, neither its preceding nor its following vehicle can engage in overtake"; this can be expressed by the following set of formulas:
   $$Sav\_p_i : init \Rightarrow AG(a\_ot_i \Rightarrow \text{NOT } a\_ot_{i-1})$$
   $$Sav\_f_i : init \Rightarrow AG(a\_ot_i \Rightarrow \text{NOT } a\_ot_{i+1})$$
   where $a\_ot_i = C4_i + C5_i + C6_i$ (vehicle in position i is in actual overtaking phase).

5.5 Results

The first abstraction we propose yields an equivalent transition relation. We just reencode the control variables of each vehicle in a more efficient way. This can be done in two manners. One is to specify an invariant restricting the domain of the control variables in such a manner that only valuations in which exactly one of the control variables is true, are considered; the second solution consists in giving an explicit reencoding by means of an abstraction relation. It consists in replacing the 12 control variables of CAR_i (encoding 12 states) by four variables. This abstraction relation is total on the "interesting" domain specified by the above mentioned invariant; it is a function from the abstract to concrete domain but not in the other direction. This has been done in order to get smaller expressions and corresponding BDDs. Therefore, the obtained abstract transition relation is bisimilar to the original one and can be used for the verification of any CTL formula. As this abstraction relation consists of independent relations concerning each CAR_i, the abstraction operator can be applied before composition. We obtain a significant reduction of memory using either the invariant or the above defined equivalent abstraction on each process CAR_i. The gain of defining an abstraction over defining an invariant is only little more than 15% in terms of numbers of nodes but nearly 40% of execution time (this gain must be interpreted carefully as the overall execution time is in both cases less than 1 minute and it is difficult to evaluate the fixed time requirement for loading and writing the files, etc).

As already mentioned, for the verification of deadlock freedom, abstraction is not possible as it is not a formula of ACTL; we tried to verify it on the concrete program constructed using the invariant as well as on the above defined equivalent abstraction. It was even not possible to evaluate the predicate *sink'* (which is differs from *sink* by the exclusion of some "useless" actions). We also tried to use a stronger invariant, where the M and OT-addresses of each vehicle are identical (this is not an abstraction, but a restriction which preserves the predicate *deadlock*) but it turns out that in this particular case this stronger invariant gives not rise to a smaller representation of the program and it does also not allow to evaluate *sink'*. From this we conclude that in some cases it may be very useful to have at least some upper approximation of the set of reachable states; this may be obtained by doing a forward analysis on some carefully chosen abstraction (not yet implemented). An upper approximation of the set of reachable states is certainly also useful in order to reduce the time and memory requirement of evaluations of fixed point formulas.

A stronger abstraction has been used in order to verify the safety properties. E.g. for the verification of Sav_p_2 we have defined an abstraction, completely abstracting away all the control variables of CAR_3 and CAR_4, we introduced a single state "server" in CAR_2 instead of the 5 server states, and in CAR_1 we grouped together client states in actual overtaking and client states not in actual overtaking; we also abstracted away the variables of the timers and the environment, which should not change the safety properties. Here we obtain for the composed system some 60% of gain in terms of number of nodes with respect to the smallest unabstracted system. Here we have a more significant gain of execution time for the compositional instead of the global abstraction (40 seconds instead of 2.50 minutes).

On this abstraction it took less than 1 minute to evaluate the property, whereas it was not possible to evaluate it on the unabstracted program. In fact, the gain of memory is not very important in this case because we have not yet introduced renaming: many of the guarded commands represent identical transition relations, but have different labels and can therefore not be grouped together into a single command. In this example this will certainly allow some gain of memory, and particularly an important gain of execution time of the evaluation of formulas as actually the same transformations are computed many times.

References

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language Lotos. *ISDN*, 14(1):25–29, 1988.

[BBG*93] A. Bouajjani, S. Bensalem, S. Graf, C. Loiseaux, and J. Sifakis. *Property Preserving Abstractions for the Verification of Concurrent Systems*. Research Report Spectre C - 40, LGI/IMAG, Grenoble, 1993. submitted to Formal Methods in System Design.

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *CAV'92, Montréal*, To appear in LNCS, 1992.

[BD92] A. Bouali and R. DeSimone. Symbolic bisimulation minimisation. In *CAV92*, To appear in LNCS, 1992.

[BFH90] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In *CAV'90*, LNCS 531, 1990.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. on Computation*, 35 (8), 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL*, 1977.

[CES83] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification: a practical approach. ACM TOPLAS, 8(2):244–263, 1986.

[CGL91] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Symposium on Principles of Programming Languages (POPL)*, ACM, 1991.

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

[EFJ90] P. Ernberg, L. Fredlund, and B. Jonsson. *Specification and Validation of a Simple Overtaking Protocol using LOTOS*. Technical Report T90006, SICS, Sweden, 1990.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *CAV'91, Aalborg*, LNCS 575, 1991.

[Fer90] J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), 1990.

[GL92] S. Graf and C. Loiseaux. *Program Verification using compositional Abstraction*. FASE/TAPSOFT 93.

[GS90] H. Garavel and J. Sifakis. Compilation and verification of Lotos specifications. In *Symposium on Protocol Specification, Testing and Verification (Ottawa)*, IFIP, North Holland, Amsterdam, 1990.

[Hoa84] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1984.

[KK86] J. Katzenelson and B. Kurshan. S/R: A Language for Specifying Protocols and other Coordinating Processes. In *5th Ann. Int'l Phoenix Conf. Comput. Commun.*, 1986.

[Koz83] D. Kozen. Results on the propositional μ-calculus. In *Theoretical Computer Science*, North-Holland, 1983.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. In *REX Workshop on Stepwise Refinement of Distributed Systems, Mook*, LNCS 430, 1989.

[Lam91] L. Lamport. *The Temporal Logic of Actions*. Technical Report 79, DEC, Systems Research Center, 1991.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. Second Int. Joint Conf. on Artificial Intelligence*, BCS, 1971.

[Mil80] R. Milner. A calculus of communication systems. In *LNCS 92*, Springer Verlag, 1980.

[Mil83] R. Milner. A calculus for Synchrony and Asynchrony. *TCS*, 25, 1983.

[Pnu86] A. Pnueli. Application of temporal logic to specification and verification of reactive systems: a survey of current trends. In *Current trends in Concurrency, Nordwijkerhout*, LNCS 224, 1986.

[Rat92] Ch. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE : Le système LESAR*. Thesis, Université J. Fourier, Grenoble, 1992.

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. *Int. Symp. Protocol Spec. Testing and Validation*, 1987.

[Sif83] J. Sifakis. Property preserving homomorphisms of transition systems. In *4th Workshop on Logics of Programs, Pittsburgh*, LNCS 164, 1983.