Symbolic Verification of Parallel Programs

Isabelle Vernier
Laboratoire IBP-MASI
Université Paris 6
4, place Jussieu, 75252 Paris - Cedex 05, France
email : isabelle.vernier@masi.ibp.fr
tel : (33 1) 44 27 71 04, fax : (33 1) 44 27 62 86

Abstract: We propose a solution to solve part of the problem of the verification of properties of programs with a finite but unknown number of processes. We consider a subclass of parallel programs modeled with Petri nets. The originality of our method is to build a symbolic graph. The nodes of the graph are predicates that represent sets of states. Properties expressed in the branching temporal logic CTL-X are verified on this graph. Our verification method presents failure cases. This is not surprising because the problem is undecidable in the general case. If no failure cases are encountered, the symbolic graph represents all the reachable states and sequences of actions of programs instantiated with a number of processes greater than a bound. The building algorithm computes this bound.

I. Introduction

The verification of parallel programs requires the specification of the programs and their representation in a formal model. The problem is to have a model with a sufficient expressive power and yet simple enough to allow the development of verification algorithms. This problem is more acute when we consider programs involving a finite but unknown number of processes. The verification of properties of such programs without fixing the parameter is undecidable [K.R. Apt, et al., 1986], [I. Suzuki, 1988]. Therefore, the proposed methods can only solve parts of the problem and have limits.

A first limit is that the proposed methods are not programmable. This is the case for the methods proposed in {Clarke, 1987 #2}, {Kurshan, 1989 #73} and {Wolper, 1989 #17}. In these three methods, the user must define a program, independent of the parameter, representative of the behaviors of the instantiated programs, i.e., programs with a fixed number of processes. In the methods of Kurshan and Wolper the verification of the representativeness of a program is programmable. If the representative program verifies a property, the instantiated programs verify it too. The representative program may have more behaviors than the instantiated programs. In the method of Clarke, the verification of the representativeness is not programmable. Once this representativeness is proved, the verification of properties of the representative program is equivalent to the verification of properties of all the programs instantiated with a large enough number of processes. This bound is defined with the representative program.

A second limit is the semi-decidability of the methods. The methods proposed in {Rho, 1993 #86} and {Balarin, #43} are of that kind. They restrict the set of programs that are considered by the methods of Kurshan and Wolper. The authors have defined necessary and sufficient conditions for the existence of a representative program. The verification of these conditions is semi-decidable. If they are verified, the algorithm gives the representative program.

A third limit is a restriction on the verified properties. In {German, 1992 #84}, the proposed method is automatic but only linear temporal logic formulas that refer to the behavior of a single process can be verified. This is a strong restriction when programs with several processes are considered.

A last limit of the existing solutions is the possibility of failures. The solution proposed in {Shtadler, 1989 #34} is of that kind. The authors give a programmable method to verify that all the instantiated programs have the same behavior regarding a set of properties expressed as linear temporal logic formulas. Two results can be achieved: the equivalence is proved or the method fails.

We propose a partial solution to the automatic verification of the behavior of programs parameterized by the number of processes. Our method allows us to verify properties and to solve the accessibility problem, which is not possible with the previous methods. We build a symbolic graph that represents all the possible behaviors and reachable states of all the programs instantiated with a sufficient number of processes. The bound is computed automatically. The verification of the properties is achieved on the symbolic graph. The work presented in this paper is a generalization and a development of the method suggested in [I. Vernier, 1994b]. We consider a more general model of parallel programs and we develop and adapt the verification algorithms that were put forward.

Our method is an extension of the methods of German and Shatdler: we propose an automatic verification method and we express the properties with branching temporal logic formulas. We consider a branching logic since it allows to express the indeterminism of parallel programs. That is not possible with a linear logic [A. Pnueli, 1985], [E.A. Emerson, et al., 1986]. The method we propose has the same limit that the method of Shatdler. Some failure cases are detected by the verification algorithm.

Section 2 presents the programs we study and the model we use. We choose Petri nets to model the programs because of their well-known expressive power [W. Reisig, 1985]. Section 3 defines the predicates that represent sets of states of the instantiated programs and that are the nodes of the symbolic graph. Section 4 gives the algorithm that builds the symbolic graph. We establish a theorem that proves that the symbolic graph represents the state graphs of the instantiated programs. Section 5 explains how properties can be verified on the graph. We give a theorem proving that the sets of states that verify a formula, computed by the verification algorithm, represent exactly the states of the instantiated programs that verify the formula. The proofs of the properties and theorems presented in this paper can be found in [I. Vernier, 1994a] . In Section 6 we conclude and present possible extensions of our work.

II. Programs and Model

The problem to solve is complex. Therefore, we have developed our method on a restricted class of programs. Once the theoretical bases are fixed, it becomes possible to study extensions of this class. We consider programs composed of two parts: n identical processes communicating through shared variables and a unique control process that manages the accesses to the shared variables. The behavior of the identical processes is independent of their identity.

We consider two extensions of this basic set. This first one, presented in [I. Vernier, 1993], models programs that need the dynamic specialization of the behavior of some processes independently of their identity. The second one allows rendez-vous between two processes. This set of programs is the one considered in the methods of Clarke, Kurshan, Wolper and German. We consider here this second model.

II.1. Petri nets

The model we have chosen to represent the parallel programs is the Petri net model. A Petri net is a bipartite graph. The vertices are the places and the transitions. The places, the transitions and the tokens represent respectively the states, the actions and the processes or variables. In the following, we do not differentiate place and state, transition and action, tokens and processes. The state of a process is given by the place that contains its token. The marking of a place is the number of tokens it contains. The initial marking of a Petri net associates with each place the number of processes or resources it initially contains. A Petri net with an initial marking is a marked Petri net.

An arc, weighted by n, that links a place p to a transition t means that n processes in state p are necessary for the firing of transition t, i.e., the execution of the action t. We say that p is an input place of t. An arc, weighted by n, that links a transition t to a place p means that n processes reach the state p after the firing of t. We say that p is an output place of t. Inhibitor arcs link places and transitions too. An inhibitor arc, weighted by n, that links a place p and a transition t means that transition t can fire only if there are less than n processes in state p. The evaluation of the condition represented by the inhibitor arcs is a task of the control process. The marking reached from another, after the firing of a transition, is obtained by decreasing marking of the input places and by increasing the marking of the output places. The value of the modification is equal to the weight of the arc that links the place and the transition. This firing rule allows to compute, from the initial marking, the state graph of the Petri net.

Graphically, circles, rectangles, arcs with an arrow and circle-headed arcs represent respectively the places, the transitions, the standard arcs and the inhibitor arcs. The weight of an arc is represented only if it is different from one. The arcs with a weight equal to zero are not represented.

We now give the formal definition of a Petri net with inhibitor arcs:

Definition 1 Petri net

A Petri net PN = <P, T, Pre, Post, Inh> is a Petri net where:

- $P = \{p_1, \ldots, p_n\}$ is a set of places, T is a set of transitions, $T \cap P = \emptyset$, $T \cup P \neq \emptyset$,

- Pre is an application from $P \times T$ to $\mathbb{N}$, that represents the weight of the arcs linking the places to the transitions,

- Post is an application from $P \times T$ to $\mathbb{N}$ that represents the weight of the arcs linking the transitions to the places,

- Inh is an application from $P \times T$ to $\mathbb{N}$ that represents the weight of the inhibitor arcs.

A state machine is a Petri net for which each transition has exactly one input place and one output place. All the arcs are weighted by one. Therefore, a transition represents an action executed by one process. The input (resp. output) place is the state of the process before (resp. after) the execution.

An extended state machine is a state machine that has inhibitor arcs and that allows one to model rendez-vous between two processes. A transition of an extended state machine has two input places at most.

Definition 2 Extended state machine

An extended state machine SM is a Petri net <P, T, Pre, Post, Inh> where:

- $\forall t \in T, \sum_{p \in P} \text{Pre}(p,t) = \sum_{p \in P} \text{Post}(p,t)$ and $\sum_{p \in P} \text{Pre}(p,t) \leq 2$.

II.2. Programs

An extended state machine models the behavior of the identical processes. A bounded Petri net models the behavior of the control process and the shared variables. The transitions that represent accesses to the variables are represented in the two Petri nets and are synchronization transitions. They are actions that are executed by one or two identical processes and the control process.

Graphically, each of these transitions is represented only once. The synchronization transition has as input (resp. output) places the input (resp. output) places of the transition in the extended state machine and in the bounded Petri net.

Example Readers and writers

We consider the problem of readers and writers. Two sets of processes are defined, the readers and the writers.

Figure 1 represents the model of this readers and writers problem. The control process is only represented by inhibitor arcs.

The readers and the writers want to access to the same resource. A writer can write only if nobody is reading or writing. A reader can read only if nobody is writing.

The possible states of the readers are: waiting_to_read and reading. Those of the writers are: waiting_to_write and writing. The actions are begin_to_read, stop_reading, begin_to_write and stop_writing.

The action begin_to_read is executable only if the number of processes that are writing equals zero. Therefore, an inhibitor arc, weighted by one, links the place writing and the transition begin_to_read. The action begin_to_write is executable only if the number of processes that are reading or writing equals zero. Therefore, two inhibitor arcs, weighted by one, link the places writing and reading and the transition begin_to_write.

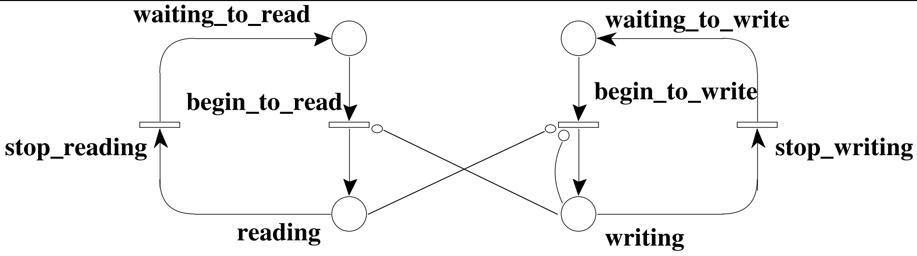

Figure 1: readers and writers model

The formula (waiting_to_read = 0) or ∃[(waiting_to_read ≥ 1) U (reading ≥ 1)] expresses the property that if a process wants to read then it is possible that a process will eventually read. The (waiting_to_read = 0) or ∀[(waiting_to_read ≥ 1) U (reading ≥ 1)] expresses the property that if a process wants to read then whatever happens that a process will eventually read.

III. Predicates

If the initial marking of a marked Petri net is modify, its state graph has to be computed again. We cannot build a state graph for each possible initial marking. We propose algorithm to build a unique symbolic graph that represents the behaviors of a Petri net with an infinity of initial markings. In this case initial marking is parameterized, i.e., the number of processes is unknown.

To represent states of the model, when the marking is parameterized, we introduce predicates that represent an aggregation of states of instantiated programs. The marking of some places may be dependent of the parameter value. The predicates establish the minimal number of tokens contained by these places. The predicates express conditions on the number of tokens contained by each place of the model. As the only parameter is the number of processes and as the number of resources is bounded, the marking of the places of the control net is independent of the parameter value.

III.1. Definitions

We introduce a function needed for the definition of the predicates. The function associates with each place the maximum value to which we compare the marking of the place during the identification of the enabled transitions.

Definition 3 Limit function

Let p be a place of the model, Limit(p) = Max(Pre(p,t),Inh(p,t)), $t \in T$.

We give two constraints to the predicates:

- There must be at least one place whose marking depends on the parameter. Otherwise, we know the value of the parameter and the predicate represents a state of one unique instantiated program.

- If we do not know the exact number of tokens in a place, we only know the minimum number of tokens in that place. We impose that this minimum value is equal or greater than Limit(p). We do not represent by a single predicate the states where more than Limit(p) tokens are in place p and those where less than Limit(p) tokens are in p. This use of the function Limit implies that the same actions can be executed from all the states represented by one predicate.

Definition 4 Predicate

Let $P = \{p_1, \ldots, p_n\}$ be the set of places of the model, a predicate is an n-tuple that defines, for each place of P, the minimal or the exact number of tokens it contains.

- predicate = $\{p_1\ op_1\ n_1, \ldots, p_n\ op_n\ n_n\}$ where $op_{i,\ i=1,\ldots,n} \in \{=, \geq\}$ and $n_{i,\ i=1,\ldots,n} \in \mathbb{N}$,

- $\exists i$, $1 \leq i \leq n$, op_i = "≥",

- $\forall i$, $1 \leq i \leq n$, if $n_i <$ Limit(p_i) then op_i = "=".

A predicate represents states of instantiated programs only if the number of processes is greater than a bound, B(predicate), equal to the sum of the n_i such that p_i is a place of the extended state machine. If the number of processes is less than this bound, it is not possible to have a distribution of processes that respects the conditions expressed by the predicate.

<u>Example</u> Predicates

We consider the model of Figure 1. For each place p of the model Limit(p) = 1.

The predicate {waiting_to_read ≥ 1, waiting_to_write ≥ 1, reading = 0, writing = 0} represents the states with at least one process is state waiting_to_read, at least one process is state waiting_to_write and no process in the states reading and writing.

The bound associated with this predicate is 2. If there is only one process, we cannot have at least one process in state waiting_to_read and at least one process in state waiting_to_write. From all the states represented by the predicate the actions begin_to_read and begin_to_write can be executed.

III.2. Expressivity of predicates

The class of programs we can model with our model is too large to make the verification problem decidable. Our method does not allow us to study all the possible models. The predicates limit the sets of states we can express. The limits are on the expression of the number of tokens in the places whose marking depends on the parameter value. We cannot represent a set of states with a place whose marking depends on the marking of another place. In the same way, we cannot represent sets of states with a place whose marking is comprised between a minimum and the parameter value but does not take all the possible values. We cannot express that the number of tokens in a place is even or odd. We will see in Section IV how we can detect that our method cannot be applied to a model.

III.3. Symbolic Firing rule

We now have to define the symbolic firing rule that allows us to determine the sets of states reachable from another set of states. The basic rule is to compute, for each predicate $\{p_1\ op_1\ n_1, \ldots, p_n\ op_n\ n_n\}$ and for each enabled transition *t*, the reachable predicate $\{p_1\ op'_1\ n'_1, \ldots, p_n\ op'_n\ n'_n\}$ as follows : $\forall i = 1, \ldots, n$ $op'_i = op_i$ and $n'_i = n_i\text{-Pre}(p_i,t)\text{+Post}(p_i,t)$.

If op_i = "≥", $\text{Pre}(p_i,t)\text{-Post}(p_i,t) \geq 1$ and $\text{Limit}(p_i) \leq n_i \leq \text{Limit}(p_i)\text{+Pre}(p_i,t)\text{-Post}(p_i,t)$, n'_i will be equal or less than $\text{Limit}(p_i)$. Several predicates are reachable. They differentiate the cases where $n'_i = \text{Limit}(p_i)$ from the cases where $n'_i < \text{Limit}(p_i)$.

The first predicate, $op'_i = op_i$ and $n'_i = \text{Limit}(p_i)$, is reachable if the marking of p_i is greater or equal to $\text{Limit}(p_i)\text{+Pre}(p_i,t)\text{-Post}(p_i,t)$.

The other ones, op'_i = "=" and $n'_i = \text{Limit}(p_i)\text{-}\alpha$, are reachable if the marking of p_i is equal to $\text{Limit}(p_i)\text{+Pre}(p_i,t)\text{-Post}(p_i,t)\text{-}\alpha$, $(\text{Limit}(p_i)\text{+Pre}(p_i,t)\text{-Post}(p_i,t)\text{-}n_i \geq \alpha \geq 1)$ and if there exists op'_j = "≥". This last condition ensures that there is at least one place whose marking depends on the parameter value.

The following figures show the application of the special case. They illustrate the case where p_i is the only input place of *t* such that $n_i \leq \text{Limit}(p_i)\text{+Pre}(p_i,t)\text{-Post}(p_i,t)$. The reachable predicates and the conditions of the firing of the transition are represented.

$$\{p_1 = n_1, \ldots, p_i \geq n_i, \ldots, p_n = n_n\}$$

$$\downarrow \; t,\ p_i \geq Limit(p_i)+Pre(p_i,t)-Post(p_i,t)$$

$$\{p_1\ op'_1\ n'_1, \ldots, p_i \geq \text{Limit}(p_i), \ldots, p_n\ op'_n\ n'_n\}$$

Figure 2

Figure 2 shows the reachable predicate if place p_i is the only one whose marking depends on the parameter ($\forall j \neq i$, op_j = "="). Figure 3 represents the reachable predicates if the marking of another place than p_i depends on the parameter ($\exists j \neq i$, op_j = "≥").

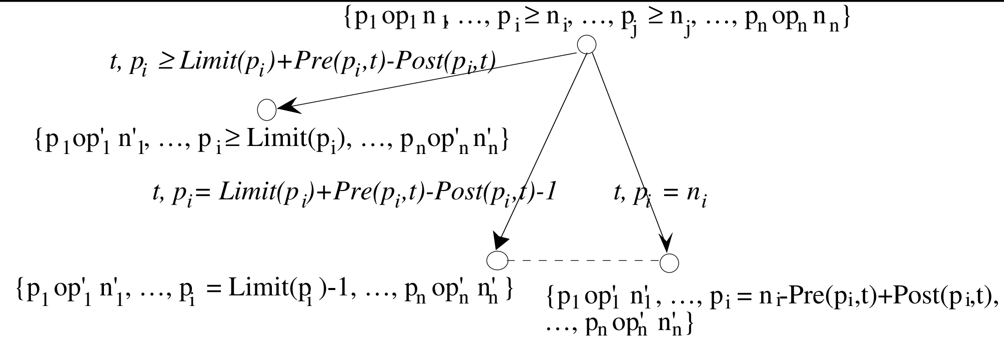

Figure 3

The same kind of reasoning is applied when t has two input places.

In Figure 2, the bound associated with the reached predicate is greater than the bound associated with the source predicate. We define a new bound, the behavioral bound (BB(predicate)), that is equal to the maximum of the bounds of the successor predicates.

Definition 5 Behavioral bound

Consider a predicate and a program instantiated with n processes:

- If $n <$ BB(predicate) then the executions from the states represented by the predicate do not represent runs of the instantiated program,
- If $n \geq$ BB(predicate) then the executions from the states represented by the predicate represent runs of the instantiated program.

The following theorem states that the firing rule we have defined is compatible with the firing rule defined for the Petri nets.

Theorem 1

Let $pred_1$ and $pred_2$ be two predicates and s_1 and s_2 two states of a program instantiated with n processes, $n \geq BB(pred_1)$,

- if a transition t is enabled in s_1 and leads to s_2, then t is enabled in all the predicates that represent s_1 and a predicate that represents s_2 is reached,
- if a transition t is enabled in $pred_1$ and leads to $pred_2$, then in all the states represented by $pred_1$ t is enabled and all the states represented by $pred_2$ are reachable, after the firing of t, from a state represented by $pred_1$.

Proof

The proof follows from the definition of the predicates and from the firing rule.

III.3. Relations on predicates

To build the symbolic graph we need to define two relations on the predicates: an inclusion relation and a superiority relation. They both apply to the sets of states represented by the predicates.

Definition 6 Inclusion

Let pred and pred' be two predicates, pred is included in pred', pred $\subseteq$ pred', if and only if the set of states represented by pred is included in the set of states represented by pred'.

Definition 7 Superiority

Let pred and pred' be two predicates, pred is less or equal to pred', pred $\leq$ pred', if and only if for each state represented by pred' there is a less state represented by pred and for each state represented by pred there is a greater state represented by pred'.

The following methods show how we test the two relations. We have proved in [I. Vernier, 1994a] that the result of the methods is equivalent to the definitions.

Let pred = $\{p_1\ op_1\ n_1, \ldots, p_n\ op_n\ n_n\}$ and pred' = $\{p_1\ op'_1\ n'_1, \ldots, p_n\ op'_n\ n'_n\}$ be two predicates, to test if pred is included in pred' we have to test that, for each index i, if op'_i = "="

then op_i = "=" and $n_i = n'_i$ else $n_i \geq n'_i$. To test that pred' is greater than pred we have to test that, for each index i, $op'_i \geq op_i$ and $n'_i \geq n_i$. We consider "≥" > "=".

IV. Symbolic Graph

We can now build a symbolic graph. We first build a tree whose nodes are predicates. To obtain the symbolic graph we identify with a node all the nodes it includes. To build the tree we use the symbolic firing rule. We have defined two new rules to avoid the building of infinite branches. The first one indicates when to stop the computation of a branch of the tree. The second one allows us to detect potentially infinite branches. Once such a branch is identified, it is possible, under some conditions, to avoid the infinite building by modifying the reached predicate. If the modification conditions are not verified, we cannot build the symbolic graph. The verification of the conditions requires the definition of a particular sequence named "state change for one process" sequence

IV.1. "State change for one process" sequence and failure cases

After the execution of a "state change for one process" sequence there is one less process in a state, the source state, and there is one more process in another state, the target state. The other processes and the resources are in the same state as before the execution. Once the sequence has been executed, it can be executed again as long as there are processes in the source state or a finite number of times that depends on the structure of the model. In the first case, the reached states represent all the possible distributions of the processes initially in the source state between the source and the target states.

Example Failure case

The Petri net of Figure 5 models the access to resources A and B by a finite but unknown number of processes. Resource A is free only if Resource B is not and conversely. A process with no resource is in state idle and a process with one resource is in state busy.

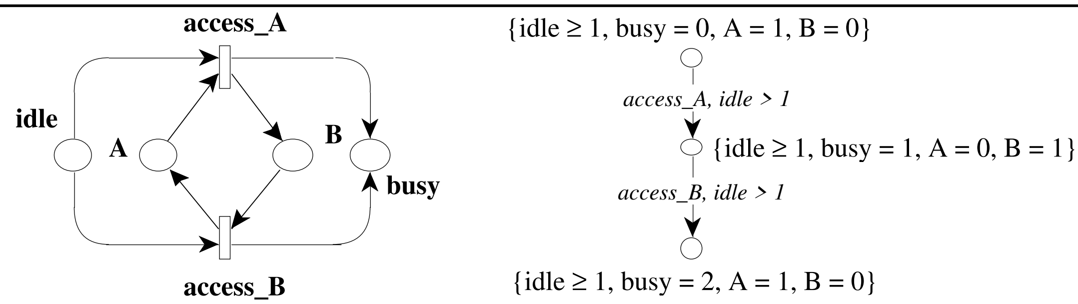

Figure 5

Initially all the processes are in the state idle and one resource A is available. This global state is represented by the predicate {idle ≥ 1, busy = 0, A = 1, B = 0}. The tree of the figure 5 shows one execution from this predicate.

The firing of transition access_A leads to the predicate {idle ≥ 1, busy = 1, A = 0, B = 1} and the firing of transition access_B leads to the predicate {idle ≥ 1, busy = 2, A = 1, B = 0}. This predicate is strictly greater than the initial one. However the sequence access_A, access_B is not a "state change for one process" sequence. After its execution two processes have moved from state idle to state busy. To avoid the building of an infinite branch we should represent, with a single predicate, all the distributions of processes between places idle and busy with an even number of processes in place busy. We cannot represent, with a finite number of predicates, such a set of states.

Definition 8 "State change for one process" sequence

Let $s = t_1 \ldots t_m$ be a sequence of transitions

Notation: We extend the applications Pre and Post to sequences of transitions. Let p be a place, $Pre(p,s) = \sum_{i=1}^{m} Pre(p,ti)$ and $Post(p,s) = \sum_{i=1}^{m} Post(p,ti)$.

s is a "state change for one process sequence" iff:

- $\exists p_{source} \in P$, $Post(p_{source},s) - Pre(p_{source},s) = -1$,
- $\exists p_{target} \in P$, $Post(p_{target},s) - Pre(p_{target},s) = 1$,
- $\forall p \neq p_{source},p_{target}$, $Post(p,s) - Pre(p,s) = 0$,
- $\forall i \leq m, \forall p \in P, \sum_{j=1}^{i} Post(p,t_j) - \sum_{j=1}^{i} Pre(p,t_j) \geq -1$,
- cond = $\forall i \leq m$, if $Post(p_{target},t_i) \geq 1$ then $Inh(p_{target},t_i) \geq 1$.

If the last condition is true, the sequence can be executed a finite number that depends on the structure of the model. Each transition of the sequence that adds processes in place p_{target} can be executed only if the number of processes in this place is less than a defined number. The number of processes in place p_{target} cannot be greater than a limit dependent on the weight of the inhibitor arcs.

IV.2. Algorithm

We can now define the rules for the building of the symbolic graph. We start from the predicate that represents the initial states of the instantiated programs and we compute the reachable predicates.

Rule 1: We use the firing rule defined in Section III.

Rule 2: If we compute a predicate that is included in one of its predecessors we stop the building of the branch. The reachable predicates are already computed.

Rule 3: If we compute a predicate that is strictly greater than one of its predecessors we examine the sequence *s* that links the two predicates. If *s* is a "state change for one process" sequence and if the last condition is true, the reached predicate is not modified. The execution of *s* will not lead to an infinite branch. If *s* is a "state change for one process" sequence and if the last condition is false, we know how to avoid the building of the infinite branch. The new predicate represents all the states reachable by executing again the sequence as long as at least one process remains in place p_{source}. It is obtained by changing in "≥" the operator associated with place p_{target}. If *s* is not a "state change for one process" sequence, we are in a failure case. We cannot represent the set of reachable states with a finite number of predicates.

Example Part of a tree

We consider the model of Figure 1. Figure 6 gives part of the tree. We do not represent the places p such that "p = 0". The names of the places and the transitions are replaced by abbreviations, the first letter of the main words that compose the name.

From the predicate {waiting_to_read ≥ 1, waiting_to_write ≥ 1} we can fire transition begin_to_read. The reached predicate is strictly greater than its predecessor. The transition begin_to_read is a "state change for one process" sequence. After its iteration, the processes initially in place waiting_to_read are distributed between places waiting_to_read and reading. All these states are represented by the new predicate {waiting_to_read ≥ 1, waiting_to_write ≥ 1, reading ≥ 1}. From this new predicate the same transition is enabled. Two predicates are reached. The first one, if the number of processes in place waiting_to_read is strictly greater than one, is included in its predecessor. We do not compute the successors of this predicate. It will be identified with its predecessor in the symbolic graph.

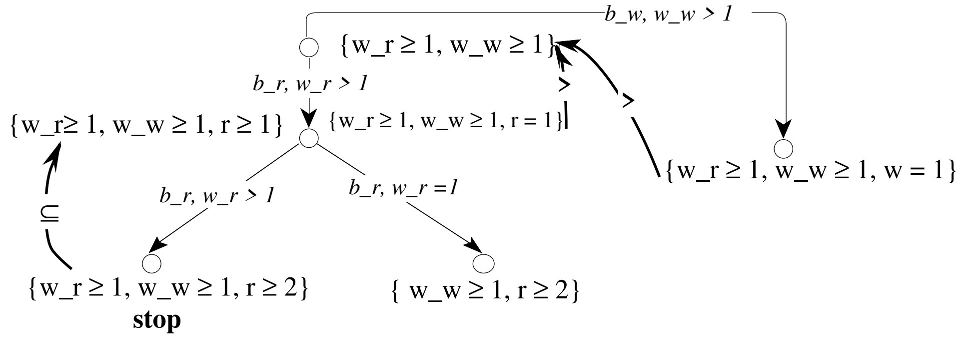

Figure 6

If we consider the action begin_to_write, that can be executed from the initial predicate, we are in the same situation. The only difference is that we cannot iterate the sequence as long as there are processes in place waiting_to_write. A process can reach the target place, writing, only if there is no process in it. Therefore, the sequence cannot be iterated. We do not modify the predicate.

Property 2

The algorithm builds a finite symbolic graph or fails.

Proof

We show that:

- if the graph is infinite, there is an infinite strictly increasing sequence of predicates,
- we cannot have an infinite strictly increasing sequence of predicates.

The proof follows from rules 2 and 3 of the building algorithm and from the finiteness of the number of places.

By instantiated programs we mean programs instantiated by numbers of processes greater or equal to the maximum of the behavioral bounds of the predicates of the graph. The following theorem ensures that the symbolic graph represents exactly the states of all the instantiated programs. All the possible executions of the instantiated programs are represented in the symbolic graph.

Theorem 2

- The symbolic graph represents all the reachable states of each instantiated program.

- Each state represented by a predicate of the symbolic graph is a reachable state of an instantiated program.

- For each sequence of actions and states of each instantiated program there is a path in the symbolic graph that represents the succession of both states and actions.

Proof

We use Theorem 1 and the fact that rule 3 computes all and only reachable states. The main difficulty is to prove that:

- the predicates computed by the application of the rule 3 represent only reachable states,

- all the states reached after the iteration of a "state change for one process" sequence are represented by the predicates computed by the application of the rule 3.

Example Symbolic graph

Figure 7 shows the symbolic graph of the model of Figure 1. The initial predicate is the black node. The graph represents the states of the programs instantiated by three processes at least.

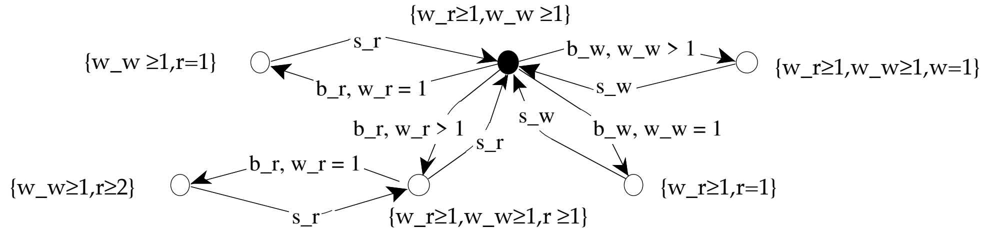

Figure 7

V. Verification of properties

Once the graph is built, we use it to verify behavioral properties of the instantiated programs. We have chosen to express these properties with branching temporal logic formulas. They allow us to consider the main properties of parallel programs and to express the indeterminism specific to parallel programs. As we prefer to avoid the formulas that allow us to count the number of processes, we restrict CTL [E.M. Clarke, et al., 1986] to CTL-X. The formulas are built as CTL formulas except that the operator X is forbidden.

If f and g are CTL-X formulas then $\neg f$, $f \wedge g$, $f \vee g$, $\forall[f\ U\ g]$ and $\exists[f\ U\ g]$ also. The interpretation of the classical operators is as usual. A state *s* verifies $\forall[f\ U\ g]$ iff for any path starting from *s* f is true on the path until g becomes true and g will eventually become true. A state *s* verifies $\exists[f\ U\ g]$ iff there is at least one path starting from *s* such that f is true on the path until g becomes true and g will eventually become true.

The classical model-checker algorithms suppose that a node of the graph is a state of the program. This is not the case with the symbolic graph. A node is a set of states that may not verify the same properties. The method we propose computes for each predicate and each formula a verification predicate that corresponds to the states represented by the original predicate that verify the formula. We verify a formula after its sub-formulas. A verification predicate is a predicate that defines set of states that verify a formula. A predicate is a node of the symbolic graph.

The verification predicates are extension of the predicates. The number of processes in each place can be exactly known. In this case, the formula is only verified by states of one instantiated program. The basic formulas are verification predicates.

The computation of the verification predicates presents no problem if the formulas do not contain temporal operators. The logical operators are successively applied to the verification predicates. We now explain our method when the formulas contain temporal operators.

<u>Evaluation of $\exists[f\ U\ g]$</u>

The principle of the evaluation of $\exists[f\ U\ g]$ is to compute verification predicates that represent sets of states that verify f and that are predecessors of states that are already known to verify the formula $\exists[f\ U\ g]$. These new states verify $\exists[f\ U\ g]$. The inspection starts from verification predicates that represent states that verify g. The verification predicates are computed by back traversal of the symbolic graph. To detect the end of the verification we apply the same conditions as for the building of the symbolic graph. The same failures may arise. The sets of states that verify the formula cannot be represented by a finite number of verification predicates.

<u>Evaluation of $\forall[f\ U\ g]$</u>

The verification of the formulas $\forall[f\ U\ g]$ requires the detection of some circuits in the graph. Our problem is that some circuits that appear in the symbolic graph represent no circuit of the state graphs of the instantiated programs. We have no efficient solution to avoid these circuits. The computation of all the circuits of the graph is too much complex. Therefore, we use heuristics.

A state verifies the formula $\neg\forall[f\ U\ g]$ if it is the source of a circuit of states that verify $f \wedge \neg g$ or if a state that verifies $\neg f \wedge \neg g$ is reachable from it and all the intermediate states verify $f \wedge \neg g$. The states of the first kind compose the set Circuit($f \wedge \neg g$) and the states of the second kind verify the formula $\exists[(f \wedge \neg g)\ U\ (\neg f \wedge \neg g)]$. If Circuit($f \wedge \neg g$) is an empty set we know that a state verifies $\forall[f\ U\ g]$ if and only if it verifies $\neg\exists[(f \wedge \neg g)\ U\ (\neg f \wedge \neg g)]$. We know how to evaluate this formula. If Circuit(f $\wedge \neg$g) is not an empty set, we do not know how to evaluate the formula.

The evaluation of the emptiness of Circuit(f $\wedge \neg$g) is also implemented with heuristics. For each predicate Π of the symbolic graph our algorithm performs the following steps:

step 1: Π' = sets of states represented by Π that verify f $\wedge \neg$g,

step 2: we build the graph G of the states that verify f $\wedge \neg$g reachable from states represented by Π',

step 3: if a state represented by Π' belongs to Circuit(f $\wedge \neg$g), the circuit is represented in G. If some circuits in G may be circuits of the state graphs of the instantiated programs Circuit(f $\wedge \neg$g) may be not empty, our heuristics fails. To test if $\Pi' \cap$ Circuit(f $\wedge \neg$g) is an empty set, we compute a set of elementary cycles of the graph and we verify if there exists a sequence composed of elementary cycles whose execution does not modify the global state of the program. This verification comes down to test if a linear system of equations has a positive integer solution. This heuristics increases the number of failure cases but gives a result that is sure.

If the three steps succeed for all the nodes of the symbolic graph we are sure that Circuit(f $\wedge$ $\neg$g) is an empty set. We evaluate the formula $\neg\exists$[(f $\wedge \neg$g) U ($\neg$f $\wedge \neg$g)] that contains one universal quantifier less.

The following theorem ensures that the results of our verification algorithm are equivalent to the results obtained on the state graphs of the instantiated programs if no failure cases are encountered.

Theorem 3

- The verification ends or fails in a finite time.

- A state of an instantiated program verifies a formula if and only if it is represented by a verification predicate that represents states that verify the formula.

Proof

The proof follows from the verification algorithm and from the fact that the algorithm includes the detection of the cases where it has not enough information to conclude.

VI. Conclusion

We have given an automatic method to verify properties of programs without specifying the number of processes. A main advantage of our method is that it deals with properties expressed in a branching time temporal logic while the automatic methods already proposed in [Z. Shtadler, et al., 1989] and [S. German, et al., 1992] only work with properties expressed in a linear time temporal logic. Another advantage is that our symbolic graph allows us to answer to the reachability problem. By inspecting all the predicates of the symbolic graph we can say if a state of an instantiated program is reachable.

The algorithm to build the symbolic graph is implemented in the software engineering framework AMI [CPN-AMI, 1994]. The user enters his model, a Petri net, through a user interface. He can call other analysis tools whose results are displayed on the interface and the environment allows tools collaboration. We have to finish the implementation of the verification algorithms.

Our method presents failure cases that are in part due to the undecidability of the problem in the general case. In the methods of Clarke, Kurshan and Wolper the existence of a representative program is not decidable, in the methods of Rho and Balarin the computation of a representative program is semi-decidable and the method of Shtadler may fails. For all these methods no class of models that can be studied with success is given. We have not identify a class of programs and of properties whose study, with our method, does not fail. To characterize the class of models we can study, it is reasonable to first consider a model less expressive than Petri nets. Once all the method implemented, it will be possible to test lot of models and to identify more precisely failure cases.

Another improvement of our work will be the consideration of more complex languages such as CTL*, FCTL [E.A. Emerson, et al., 1985]. The model-checkers that work with such languages compute the strongly connected components of the state graph. Therefore we will be able to deal with these logics if we can compute, from the symbolic graph, strongly connected components that represent the ones of the state graphs of the instantiated programs.

References

[Apt K.R., et al., 1986] Apt K.R., Kozen D.C., "*Limits for automatic verification of finite-state concurrent systems*", Information Processing Letters, vol 22:6, pp 307-309, 1986.

[Balarin F., et al.,] "*On the Automatic Computation of Network Invariants*", unpublished.

[Clarke E.M., et al., 1992] Clarke E.M., Burch J.R., Grumberg O., Long D.E., McMillan K.L., "*Automatic verification of sequential circuit designs*", Philosophical Transactions of the Royal Society of London, Series A, vol 339, pp 105-120, 1992.

[Clarke E.M., et al., 1986] Clarke E.M., Emerson E.A., Sistla J., "*Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification*", ACM Transactions on Programming languages and systems, vol 8, n°2, pp 244-263, 1986.

[CPN-AMI, 1994] CPN-AMI, "*Reference manual, 1.2 version*", 1994.

[Emerson E.A., et al., 1986] Emerson E.A., Halpern J.Y., "*'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time*", Journal of ACM, vol 33, n°1, pp 151-178, 1986.

[German S., et al., 1992] German S., Sistla A.P., "*Reasoning about Systems with Many Processes*", JACM, vol 39, pp 675-735, 1992.

[Kurshan R.P., et al., 1989] Kurshan R.P., McMillan K., "*A Structural Induction Theorem for Processes*", Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Alberta, Canada, pp 239-247, 1989.

[Pnueli A., 1985] Pnueli A., "*Linear and branching structures in the semantics and logics of reactive systems*", Proceedings of the 12th International Colloquium on Automata, Languages and Programming, Springer Verlag, pp 15-32, 1985.

[Reisig W., 1985] Reisig W., EATCS Monographs on Theoretical Computer Science, Springer Publ. Company, vol 4, 1985.

[Rho J.K., et al., 1993] Rho J.K., Somenzi F., "*Automatic generation of networks invariants for the verification of iterative sequential systems*", Proceedings of the 5th International Conference, CAV'93, Elounda, Greece, LNCS 697, pp 123-137, 1993.

[Shtadler Z., et al., 1989] Shtadler Z., Grumberg O., "*Network Grammars, Communication Behaviors and Automatic Verification*", Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, LNCS 407, pp 152-165, 1989.

[Suzuki I., 1988] Suzuki I., "*Proving properties of a ring of finite-state machines*", Information Processing Letters, vol 28, n°4, pp 213-214, 1988.

[Vernier I., 1993] Vernier I., "*Specification And Verification of Parametrized Parallel Programs*", Proceedings of the 8th International Symposium on Computer and Information Sciences, Istanbul, Turkey, pp 622-625, 1993.

[Vernier I., 1994a] Vernier I., "*Graphe symbolique paramétré de réseaux de Petri et Logique temporelle*", Thèse d'Université, Université Paris 6, 1994.

[Vernier I., 1994b] Vernier I., "*Parameterized Evaluation of CTL-X Formulae*", Proceedings of the 1st International Conference on Temporal Logic Workshop, Bonn, Germany, Max-Planck-Institut Für Informatik technical report 94-230, pp 22-31, 1994.

[Wolper P., et al., 1989] Wolper P., Lovinfosse V., "*Verifying Properties of Large Sets of Processes with Network Invariants*", Proceedings of the Proc. International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, LNCS 407, pp 68-80, 1989.