Symbolic Executions of Symmetrical Parallel Programs

Also in the proceedings of of 4th Euromicro Workshop on Parallel and Distributed Processing, Braga, Portugal, Janvier 1996, pages 327-334.

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 an algorithm to execute symbolically parallel programs with an unknown number of processes. Usually one instantiates the programs before their execution, i.e., one fixes the number of processes. We exploit the symmetrical behaviors of the processes to execute simultaneously all the instantiated programs. We define symbolic states that represent sets of states of an infinite number of instantiated programs. We compute simultaneously all the successors of states represented by a symbolic one. After an exhaustive execution, we build a graph that represents all the reachable states and possible executions of the instantiated programs. We can explore this symbolic graph to verify properties expressed with temporal logic formulas.

I. Introduction

Many parallel programs are described with a parameter, the number of processes, and are instantiated, i.e., the number of processes is fixed, to be executed. It is usually impossible to extend to all the instantiated programs the results of the analyze of a particular one. It is prohibitive to analyze again a program for multiple values of the parameter. A way to overcome this drawback is to analyze a program independently of the value of the parameter. We present such a method.

Formal methods exist to verify properties of parameterized programs even if the number of processes is unknown. They all work on symmetrical parallel programs. Processes have symmetrical behaviors if they have the same possible executions that are independent of their identity [2]. Symmetric algorithms as termination detection [7], mutual exclusion [6, 10] or reaching agreement algorithms [11], compose an important subclass of the parallel algorithms. They may be categorized according to two criteria [12]: the knowledge that a process has of its environment and the influence of a process on the behavior of the others. In our work, we consider that processes do not know about their identity and the number of other processes. They still are aware of the presence of the other processes. Processes communicate through shared variables or directly with other ones. All the works on parameterized programs assume the same hypothesis.

As the general problem is undecidable, methods concern only part of it. The principle common to the methods is to find an abstraction of the instantiated programs, independent of the number of processes, that preserves a given set of properties. The abstraction verifies a property of the set if and only if all the instantiated programs do so. Once the abstraction is defined, one builds its state graph and checks properties expressed with temporal logic formulas. All the methods give a solution only if the instantiated programs verify the same properties for all the number of processes greater than a bound. The programs instantiated with fewer processes may have a particular behavior. The problem is to find the abstraction. One cannot always decide the existence of an abstraction. It depends on the set of possible abstractions we consider. In [15] the set is finite therefore the method fails if it does not find one abstraction in the set. In [1, 4, 9, 14, 19], the set of possible abstractions is not limited. The methods that are described in [4, 9, 19] are not fully automated. In [8], only properties that refer to the behavior of a single process can be verified.

All these methods do not allow to represent reachable states and execution sequences of the instantiated programs. They give no intermediate results if it is impossible to build the state graph of the abstraction. The size of the state graph may be too big or infinite. We propose a method to execute symbolically parameterized parallel programs. We can build a symbolic graph and explore it to verify properties expressed with temporal logic formulas when we have computed all the possible executions and reachable states. In [18] we present the verification algorithm. Our method allows us to compute the bound that separates the instantiated programs with a particular behavior from the others.

The main point is to represent, with a symbolic state, set of states of instantiated programs. We bring states together according to the actions they enable and the states where the processes and resources are. For each action we compute the successor symbolic states of another one at the same time. When it is not possible to represent with a finite number of symbolic states all the states of the instantiated programs, the method fails. Otherwise, the finite symbolic graph represents exactly all the reachable states and executions of all the instantiated programs. We give the proofs of both properties and theorems in [17].

Section 2 presents the programs we study and the model we use. We choose extended Petri nets to model the programs because of their well-known expressive power [13]. The symbolic states are symbolic markings of the Petri nets. Section 3 defines the symbolic markings. Section 4 gives the rule to execute the Petri net and conditions to have an exhaustive execution. In Section 5 we conclude and present possible extensions of our work.

II. Symmetrical parallel programs and formal model

Two parts including a parameterized one compose the programs we consider. The parameterized part represents the behavior of *n* identical processes that communicate through shared variables or by synchronization. The other part, the control net, concerns the processes in given number and resources. It is an optional part. The control net manages the accesses to the resources. The conditions that govern the execution of each action are independent of the identity of the processes.

II.1. Petri nets

We model parallel programs by means of Petri nets. A Petri net is a bipartite graph. The vertices are the places and the transitions. The places and the transitions represent respectively the states or values and the actions. A token represents a process or a variable. The place that contains the token associated with a process (or a variable) gives its state (or its value). The marking of a place is the number of tokens it contains. The marking of a net is a vector. It associates with each place its marking. The initial marking represents the initial state of the program. The firing of a transition corresponds to the execution of an action.

An arc weighted by *n*, that links a place *p* to a transition *t*, *p* is an input place of *t*, means that *n* tokens in *p* are necessary for the firing of *t*. An arc, weighted by *n*, that links a transition *t* to a place *p*, *p* is an output place of *t*, means that *n* processes reach *p* after the firing of *t*.

Graphically, circles, rectangles and arcs with an arrow represent respectively the places, the transitions and the arcs. We represent the weight of an arc only if it is different from one. We do not represent the arcs with a weight that equals zero.

Definition 1 Petri net and marking

A Petri net is noted <P, T, Pre, Post> where:

- P is a set of places, T is a set of transitions, $T \cap P = \{\}$, $T \cup P \neq \{\}$,
- Pre, an application from $P \times T$ to $\mathbb{N}$, represents the weight of the arcs linking the places to the transitions,
- Post, an application from $P \times T$ to $\mathbb{N}$, represents the weight of the arcs linking the transitions to the places,
- a marking m is an application from P to $\mathbb{N}$.

To test the enabling of a transition, we compare for each place, the values of both the marking and the application Pre.

Definition 2 Enabling of a transition

Let P = <P, T, Pre, Post> be a Petri net, m a marking and t a transition that belongs to T. The marking m enables t, m[t>, if and only if:

$\forall P \in P$, $m(p) \geq Pre(p,t)$.

The decrease (respectively increase) of the marking of the input (respectively output) places gives the marking reached after the firing of a transition. The value of the decrease equals the weight of the arc that connects the place to the transition. The value of the increase equals the weight of the arc that connects the transition to the place. This firing rule allows one to compute the reachable markings of the Petri net.

Definition 3 Firing rule

Let P = <P, T, Pre, Post> be a Petri net, m a marking and t a transition enabled from m. The marking m' reached after the firing of t, m[t>m', is computed as follows:

$\forall p \in P$, $m'(p) = m(p) - Pre(p,t) + Post(p,t)$.

A state machine is a Petri net. Each of its transitions has exactly one input place and one output place. The weight of all the arcs is one. A transition represents an action executed by one process. The input (respectively output) place is the state of the process before (respectively after) the execution. An extended state machine allows one to model synchronization between several processes. A fixed number of processes can cooperate to execute a common action. An action does not create or kill a process.

Definition 4 Extended state machine

An extended state machine is noted <P, T, Pre, Post> where:

- <P, T, Pre, Post> is a Petri net,
- $\forall t \in T$, $\sum_{p \in P} Post(p,t) = \sum_{p \in P} Pre(p,t)$.

II.2. Programs

An extended state machine models the behavior of the identical processes. A Petri net models the behavior of the control net. P (respectively T) denotes the set of places (respectively transitions) of the whole Petri net. P_c (respectively P_p) is the set of places of the control net (respectively the extended state machine). The actions of synchronization are of two kinds: between several identical processes or between process(es) and the control manager.

Transitions common to the two nets represent synchronization actions.

Example 1 Rooms and users

Let us consider a problem of access to rooms. Several users cannot access simultaneously to the same room. All the keys open all the rooms. Petri net of Figure 1 models the system. Each user is in state idle since he decides to work. Then he is waiting for one key. Once he holds one key and if one room is free he enters it and gives the key back. To leave the room he must have a key to lock the door. He gives the key back at the same time as he leaves the room. The users are the processes.

The places of the control net are: keys and rooms. All the transitions, apart from the transition want-to-work, are synchronization transitions between the control net and one process. We fix the number of rooms and keys before computing the reachable states of the program.

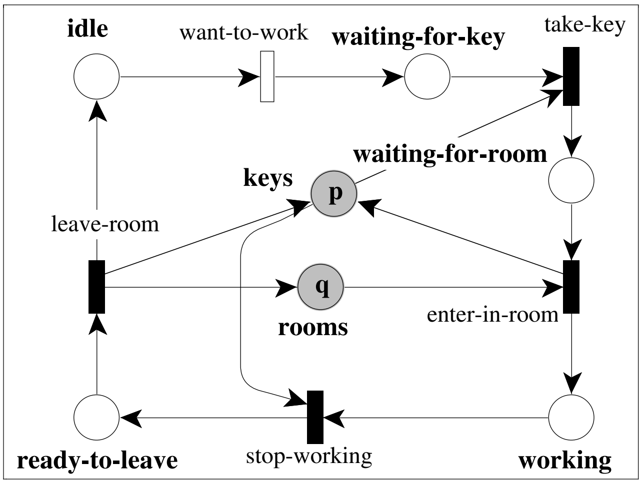

Figure 1: rooms and users model

III. Symbolic markings

Markings associate with each place an integer that represents the number of tokens it contains. They are not suitable to consider states of parameterized programs since the number of identical processes is unknown. For some places, we only know the minimum number of tokens they contain. To this purpose we define symbolic markings. They represent infinite sets of markings of instantiated programs. A symbolic marking represents markings that enable the same transitions.

III.1. Symbolic markings

The key point of the representation of a symbolic marking is to associate an integer and a relational operator "=" or "≥" with each place. The intended meaning is:

- if "= x" is associated with a place, then exactly x processes are in this place,
- if "≥ x" is associated with a place, then at least x processes are in this place.

To ensure that a symbolic marking represents only markings that enable the same transitions, we define a function Limit. It associates with each place the maximum number of tokens necessary for the firing of all the transitions. This function computes the maximum value to which we may have to compare the marking of each place.

Definition 5 Limit function

$\forall p \in P$, Limit(p) = MAX(Pre(p,t)), $t \in T$.

We use the function Limit to define symbolic markings. If we do not know the exact number of tokens in a place p the integer associated with this place must be greater or equal to Limit(p). We will explain in Section III.2 why this condition is sufficient to ensure that a symbolic marking represents only markings that enable the same transitions. As the control process is not parameterized, we associate the operator "=" with each of its places.

Definition 6 Symbolic marking

sm = <OP, M> is a symbolic marking with:
- OP is an application from P to {"=", "≥"}
- M is an application from P to $\mathbb{N}$,
- $\forall p \in P_C$, sm.OP(p) = "=",
- $\forall p \in P$, (1)
 if sm.M(p) < Limit(p) then sm.OP(p) = "=".

Notation

{p sm.OP(p) sm.M(p) / $p \in P$} denotes a symbolic marking sm. We omit the places such that sm.M(p) = 0 and sm.OP(p) = "=".

We note set[sm] the set of markings represented by sm.

Definition 7 Set of markings

A symbolic marking sm represents a marking m if an only if m fulfills the relations expressed by the functions sm.OP and sm.M. A symbolic marking sm contains a marking m, $m \in$ set[sm], if and only if:

$\forall p \in P$, if sm.OP(p) = "=" then m(p) = sm.M(p)
 else m(p) ≥ sm.M(p).

A symbolic marking represents markings of an instantiated program only if the number of identical processes is greater than a bound. This bound corresponds to the minimum number of processes necessary to respect the conditions expressed by the symbolic markings.

Definition 8 Bound associated with a symbolic marking

Let sm be a symbolic marking then $B(sm) = \sum_{p \in P_p} sm.M(p)$

Property 1

Let PP be a program instantiated with n, M the set of possible markings of PP and sm a symbolic marking, set[sm] $\cap$ M $\neq \emptyset$ if and only if $n \geq B(sm)$.

Example 2 Symbolic markings

The symbolic marking {idle $\geq$ 1, waiting-for-key $\geq$ 1, keys = p, rooms = q} represents markings of programs with at least two processes. If the program is instantiated with one process, this process cannot be in state idle and waiting-for-key simultaneously.

To symbolically execute parallel programs, it is essential to be able to compare two symbolic markings. The two relations we define, inclusion and inferiority, are partial relations. They both apply to the sets of markings represented by the symbolic markings.

Definition 9 Inclusion

Let sm and sm' be two symbolic markings, sm is included in sm', sm $\subseteq$ sm', iff:

- $\forall$m $\in$ set[sm], m $\in$ set[sm'].

Definition 10 Superiority

Let sm and sm' be two symbolic markings, sm is greater than sm', sm $\geq$ sm', iff:

- $\forall$m $\in$ set[sm], $\exists$m' $\in$ set[sm'], m $\geq$ m',
- $\forall$m' $\in$ set[sm'], $\exists$m $\in$ set[sm], m $\geq$ m'.

The algorithms to test the two relations compare the operators and integers associated with each place. We present them in [17].

III.2. Enabled transitions

We now define the enabling of a transition and we prove that a symbolic marking enables the same transition as all the markings it represents.

Definition 11 Enabling of a transition

Let sm be a symbolic marking and t a transition, sm enables t, sm[t>, iff:

$\forall$p $\in$ P, sm.M(p) $\geq$ Pre(p,t) or sm.OP(p) = "$\geq$".

The definition of a symbolic marking ensures that if sm.M(p) < Pre(p,t) then sm.OP(p) = "=". Therefore, if sm.OP(p) = "$\geq$", we are sure that sm.M(p) $\geq$ Pre(p,t) for each t. As each marking m represented by sm is such that m(p) $\geq$ sm.M(p), we obtain m(p) $\geq$ Pre(p,t). This condition ensures that a symbolic marking enables a transition only if all the markings it represents enable the transition. If sm represents a marking such that m(p) < Pre(p,t), we are sure that sm.M(p) = m(p) and sm.OP(p) = "=". Each marking m represented by sm is such that m(p) = sm.M(p). This condition ensures that if a marking represented by a symbolic one does not enable a transition, all the markings represented by the symbolic one do so.

Property 2

Let sm be a symbolic marking and t a transition:

- if $\exists$m $\in$ set[sm] and m[t> then sm[t>,
- if $\exists$m $\in$ set[sm] and m does not enable t then sm does not enable t.

IV. Symbolic execution

As all the markings represented by the same symbolic markings enable the same transitions, it is interesting to be able to study simultaneously their evolution. This is what we call symbolic execution. We do not represent the execution from a marking but from a set of markings.

IV.1. General symbolic markings

We introduce the notion of general symbolic markings to simplify the symbolic firing rule. The firing of a transition may lead to a general symbolic marking. They are less restrictive than symbolic markings. General symbolic markings gather markings regarding the places where the tokens are. They do not take into account the enabling of transitions. There is no relation between the Limit function and the operator associated with a place. We suppress the condition (1) of definition 6. We can split each symbolic marking into a finite number of disjoint symbolic markings.

Definition 12 Splitting up

Let gsm be a general symbolic marking and p a place such that gsm.M(p) < Limit(p) and gsm.OP(p) = "$\geq$". We split up gsm into gsm_0, ..., gsm_n (n = Limit(p)-gsm.M(p)) where:

- $\forall$p' $\neq$ p, $\forall$i, 0 $\leq$ i $\leq$ n, gsm_i.M(p') = gsm.M(p') and gsm_i.OP(p') = gsm.OP(p'),
- $\forall$i, 0 $\leq$ i < n, gsm_i.M(p) = gsm.M(p) + i and gsm_i.OP(p) = "=", (the number of processes in place p is less than Limit(p)),
- gsm_n.M(p) = Limit(p), gsm_n.OP(p) = "$\geq$" (the number of processes in place p is greater than Limit(p)).

We apply the same procedure to the gsm_i until we compute only symbolic markings.

Property 3

- The result of the splitting up of a general symbolic marking is unique.
- At each step of the splitting up, we compute only disjoint sets of markings.
- A general symbolic marking represents the same set of markings as the union of the sets represented by the symbolic markings obtained after the splitting up.

Example 3 Splitting up of a general symbolic marking

Let gsm = {p $\geq$ 1, q $\geq$ 1, r = 2}, Limit(p) = 2 and Limit(q) = 1. gsm.M(p) < Limit(p) and gsm.OP(p) = "$\geq$". We split up gsm into the two following symbolic markings:

sm_0 = {p = 1, q $\geq$ 1, r = 2}, sm_1 = {p $\geq$ 2, q $\geq$ 1, r = 2}.

IV.2. Symbolic firing rule

The symbolic firing rule is similar to the firing rule of Petri nets. We apply the modifications to the integers associated with the places. It may lead to a general symbolic marking.

Definition 13 Symbolic firing rule

Let sm be a symbolic marking and t a transition, sm enables t and the reached set of markings gsm', sm[t>gsm' is computed as follows:

- $\forall p \in P$,
  gsm'.M(p) = sm.M(p) - Pre(p,t) + Post(p,t),
  gsm'.OP(p) = sm.OP(p),
- gsm' is split up.

The general symbolic marking gsm' represents exactly the markings reachable from sm. The symbolic markings sm'_0 … sm'_n are the result of the splitting up of gsm'. The markings m belonging to set[sm] that lead to a marking of set[sm'_i] are such that: if sm'_i.OP(p) = "=" then m(p) = sm'_i.M(p) + Pre(p,t) - Post(p,t) else m(p) $\geq$ sm'_i.M(p) + Pre(p,t) - Post(p,t)+1.

Property 4

Let sm be a symbolic marking, gsm a general symbolic marking and t a transition. gsm is split up into sm_0, …, sm_n. If sm[t>gsm then:

- $\forall m \in$ set[sm], $\exists m'$, m[t>m' and $\exists i$, $0 \leq i \leq n$, $m' \in$ set[gsm_i],
- $\forall i$, $0 \leq i \leq n$, $\forall m' \in$ set[gsm_i], $\exists m \in$ set[sm], m[t>m'.

Example 4 Reachable symbolic markings

We consider the Petri net of Figure 1 and the symbolic marking {idle $\geq$ 1, waiting-for-key $\geq$ 1, keys = p, rooms = q}. The firing of transition want-to-work leads to the general symbolic marking {idle $\geq$ 0, waiting-for-key $\geq$ 1, keys = p, rooms = q}. It is split up into two symbolic markings represented by Figure 2. On the transitions we represent the conditions verified by the markings that lead to each component of the splitting up.

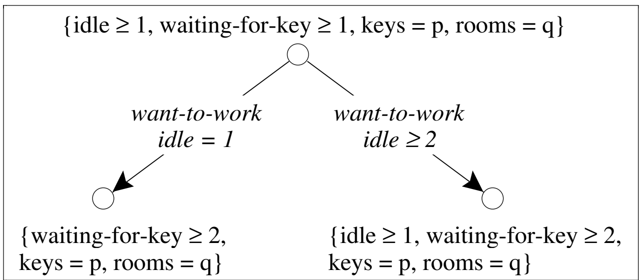

Figure 2: firing of transition want-to-work

The symbolic marking {waiting-for-key $\geq$ 2, keys = p, rooms = q} represents all the markings reached after the firing of transition want-to-work when one process is in place idle and at least one process is in place waiting-for-key. The symbolic marking {idle $\geq$ 1, waiting-for-key $\geq$ 2, keys = p, rooms = q} represents the states reached after the firing of transition want-to-work when the place idle holds at least two processes. This last execution represents an execution of programs instantiated by at least three processes.

IV.3. Exhaustive executions

During an exhaustive execution, we compute all the reachable markings. An execution may be exhaustive only if the set of reachable markings is finite. Some infinite executions are inherent in the program. The symbolic firing rule introduces some others.

Usually, the marking computed with a firing rule represents the state changes of processes and resources. With the symbolic firing rule the decrease of processes in some places may not be represented. This may introduce infinite sequences of symbolic markings. We may avoid these infinite sequences if we are able to represent it with a finite number of symbolic markings. This possibility depends on features of the sequence of transitions. We present here such sequences, the "state change for one process" sequences.

Example 5 Infinite sequence

From the symbolic marking {idle $\geq$ 1, keys = p, rooms = q} the firing of transition want-to-work leads to two symbolic markings including {idle $\geq$ 1, waiting-for-key = 1, keys = p, rooms = q}. It is greater than its predecessor. If we iterate the sequence we obtain the symbolic markings of Figure 3.

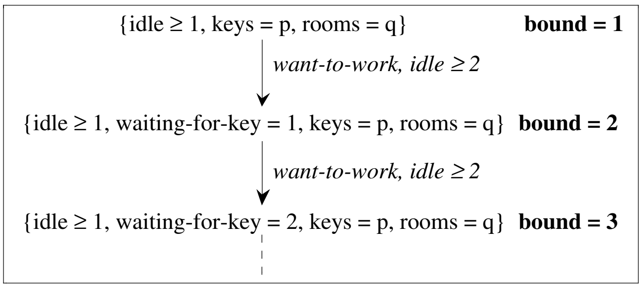

Figure 3: infinite sequence

The reached symbolic markings do not represent that after the execution of action want-to-work there is one process less in place idle. The only trace of this fact is the increase of the bound. A single symbolic marking can represent all the symbolic markings reached after the iteration of sequence want-to-work:

{idle $\geq$ 1, waiting-for-key $\geq$ 1, keys = p, rooms = q}.

A "state change for one process" sequence has the characteristic that after its execution only one process has moved from a source place to a target place. All the other processes and resources return to or stay in the same state as before the execution. If the execution of the program does not represent the decrease of the number of processes in the source state, we may have an infinite sequence of symbolic markings. Each of them differs from its predecessor in the number of processes in the target place. It is increased by one. All these symbolic markings can be represented by a single one that represents all the possible distributions of the processes initially in the source place between the source and the target places.

We now give the formal definition of a "state change for one process" sequence.

Notation We extend the applications Pre and Post to sequences of transitions. Let p be a place, $s = t_1 \ldots t_m$ a sequence of transitions.

$$Pre(p,s) = \sum_{i=1}^{m} Pre(p,t_i),\ Post(p,s) = \sum_{i=1}^{m} Post(p,t_i).$$

Definition 14 "State change for one process" sequence

Let $s = t_1 \ldots t_m$ be a sequence of transitions, s is a "state change for one process sequence" iff:

- $\exists p_{source} \in P_p$, $Post(p_{source},s) - Pre(p_{source},s) = -1$,
- $\exists p_{target} \in P_p$, $Post(p_{target},s) - Pre(p_{target},s) = 1$,
- $\forall p \neq p_{source}, p_{target}$, $p \in P$, $Post(p,s) - Pre(p,s) = 0$,
- $\forall i \leq m$, $\forall p \in P_p$,

$$\sum_{j=1}^{i} Post(p,t_j) - \sum_{j=1}^{i} Pre(p,t_j) \geq -1.$$

To execute symbolically a model, we apply the following procedure. We denote S the set of symbolic markings whose successors must be computed.

S = {initial symbolic marking}
While S is not empty do
    sm is an element of S,
    S = S - {sm},
    If sm is not included in one of its ancestors
    then
        If sm is greater than one of its ancestors
        then
            We examine the sequence that links sm to its ancestor.
            If it is not a "state change for one process" sequence
            then
                We inform the user that the execution will be infinite
            else
                If the integer associated with the place p_{target} is less than $Limit(p_{target})$
                then
                    We continue
                else
                    We modify the symbolic marking by changing to "≥" the operator associated with the target place.
        We compute the successor symbolic markings of sm and we add them to S.
    else
        The execution from sm will give no new information.

When we have no new symbolic marking to examine, we have performed an exhaustive execution. The symbolic markings that represent the reachable markings and executions of the instantiated programs are in finite number.

The following theorem ensures that the symbolic execution is correct. We give the proof in [17].

Theorem

- A symbolic marking, modified because of the execution of a "state change for one process" sequence, represents exactly all the markings, with at least $Limit(p_{source})$ processes in the source state, reachable by the iteration of the "state change for one process" sequence.
- If the symbolic execution is finite, we compute exactly all the reachable states and sequences of the instantiated programs.

Example 6 Part of a symbolic execution

The following tree represents a part of the symbolic execution of the model of Figure 1 when there are 2 keys and 3 rooms and all the processes are initially in state idle. We only consider transitions want-to-work and take-key.

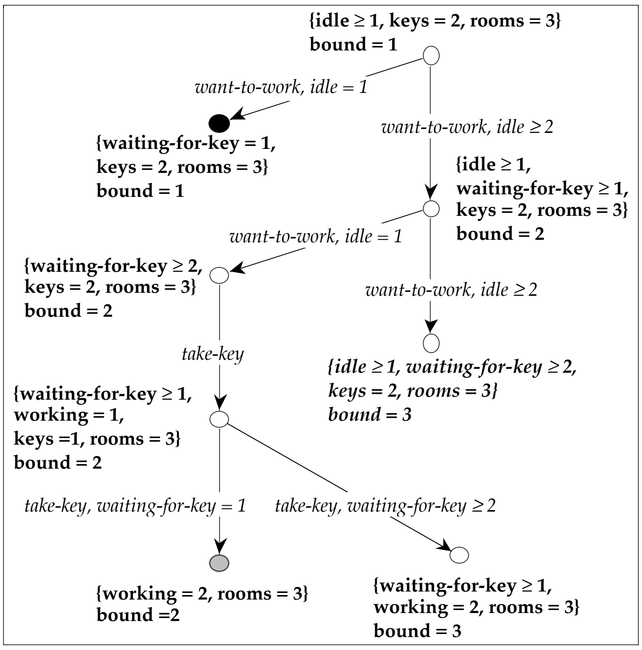

Figure 4: part of the symbolic execution

As the number of processes in each state is known, the executions from the black (respectively gray) node are executions of the programs with exactly one (respectively two) process(es). The two successors of the initial node result from the splitting up of a general symbolic marking and from the execution of a "state change for one process" sequence. The symbolic marking {idle $\geq$ 1, waiting-for-key $\geq$ 2, keys = 2, rooms = 3} is included in its predecessor. Its study will give no new results. As the bound associated with the symbolic marking {waiting-for-key $\geq$ 1, working = 2, rooms = 3} is 3, the symbolic executions from it will represent executions of programs instantiated with at least three processes. To study the programs with one or two processes, we have to compute the markings reachable from the black and gray nodes.

The symbolic execution of the model gives an exhaustive one. The whole graph gives the following information:

- programs with less than 7 processes have particular behaviors.

- with more than 4 processes the program deadlocks when 3 processes are working, 2 are waiting for a room and all the others, if any, are waiting for a key. The programs with fewer processes have no deadlock.

The whole symbolic graph has 80 states and 140 edges. The graph of the program with 7 processes has 110 states and 227 edges. To detect that a deadlock appears when the number of processes is greater than 4, if no symbolic method is used, it is necessary to build the reachability graphs of the programs with 1 to 5 processes. This will not assure that a deadlock exists for all programs with more than 7 processes. The five graphs represent 171 nodes and 373 edges.

V. Conclusion

We propose an algorithm to symbolically execute parallel programs with a finite but undefined number of processes. This algorithm allows us to execute, at the same time and by the same computations, an infinite number of instantiations of a parallel program. We have exhibit conditions that ensure that an exhaustive execution is possible. In this case, we can build a graph used for the verification of temporal logic formulas [18]. The number of computed symbolic markings is limited by the size of the memory. It will be interesting to see how to reduce the space used in particular with the BDDs and its extensions [3]. We have implemented and integrated the execution algorithm in the AMI framework [5]. The Petri nets are described through the user interface Macao.

We have identified a type of sequences that allow us to aggregate the information of the executions. Further works will deal with the definition of other types of sequences.

As an integer represents the minimum number of tokens in a place, we cannot study all the programs with the symbolic markings we have defined. For some parallel programs, the aggregation of the information requires the definition of new symbolic markings. This is the case when the possible executions depend on the parity of the number of processes or on relations between the number of processes and the number of resources. An advantage of our restriction is that the operations on the symbolic markings are easy and fully automated.

Another extension of our work will be the consideration of more complex programs. In [16] we have presented a model that allows one to distinguish dynamically the behavior of one process. We describe in [17] the adaptation of the symbolic execution algorithm.

It will be important to study the effects of the extensions of the exhaustive execution algorithms on the possibility to verify temporal logic formulas.

References

1. Balarin F., Sangiovanni-Vincentelli A.L., "*On the Automatic Computation of Network Invariants*", proceedings of Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, David L. Dill (Eds.), LNCS 818, Springer-Verlag, pp 234-246, 1994.
2. Bougé L., "*On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes*", Acta Informatica, vol. 25, pp 179-201, 1988.
3. Bryant R.E., "*Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*", ACM Computing Surveys, vol. 24, n°3, pp 293-318, September, 1992.
4. Clarke E.M., Grumberg O., "*Avoiding The State Explosion Problem in Temporal Logic Model Checking Algorithms*", Proceedings of the 6th ACM Symposium on Principles of Distributed, Vancouver, British Columbia, pp 244-303, 1987.
5. CPN-AMI, "*Reference manual, 1.2 version*"1994.
6. Dijkstra E.W., "*Solution of a Problem in Concurrent Programming Control*", Communications of the ACM, vol. 8:9, September, pp 569, 1965.
7. Dijkstra E.W., Scholten C.S., "*Termination Detection for Diffusing Computations*", Information Processing Letters, vol. 11:1, August, pp 1-4, 1980.
8. German S., Sistla A.P., "*Reasoning about Systems with Many Processes*", JACM, vol. 39, pp 675-735, 1992.
9. 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.
10. Lamport L., "*A new Solution of Dijkstra's Coucurrent Programming Problem*", Commications of ACM, vol. 17, n°8, pp 453-455, August, 1974.
11. Pease M., Shostak R., Lamport L., "*Reaching Agreement in the Presence of Faults*", Journal of ACM, vol. 27,n°2, pp 228-234, April, 1980.
12. Raynal M., "*Algorithms for Mutual Exclusion*", *Ed.* North Oxford Academic Publishers Ltd, 1986.
13. Reisig W., "*EATCS Monographs on Theoretical Computer*", *Ed.* Science, Springer Publ. Company, 1985.
14. 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, LNCS 697, Elounda, Greece, pp 123-137, 1993.

15. Shtadler Z., Grumberg O., "*Network Grammars, Communication Behaviors and Automatic Verification*", Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, Grenoble, France, pp 152-165, 1989.
16. 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.
17. Vernier I., "*Graphe symbolique paramétré de réseaux de Petri et Logique temporelle*", thèse de Université Paris 6, 1994.
18. Vernier I., "*Symbolic Verification of Parallel Programs*", IBP - MASI report, n° 95/03, 1995.
19. 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, LNCS 407, Grenoble, France, pp 68-80, 1989.