Parameterized Evaluation of CTL-X formulae

Vernier Isabelle
Labo. MASI
Université Pierre et Marie Curie
4, place Jussieu
75252 Paris Cedex 05
Email : vernier@masi.ibp.fr
Tel : (33-1) 44-27-71-04

Abstract : Temporal formulae of languages like LTL, CTL-X, CTL, … may be efficiently verified for systems described by finite automata. However these methods require to fix the different parameters of the system such as the number of processes or resources. Therefore new algorithms must be developed to study parallel programs with parameters. We present such an algorithm. The systems we study can be decomposed into two synchronized automata. One of the two automata represents the behaviour of a finite but unknown number of processes and the other one represents the control of these processes. Many systems can be modelled this way. The verification algorithm is divided into two algorithms. The first algorithm tries to build a parameterized state graph independent of the number of processes. Then, if it ends with success, the second algorithm can evaluate any formula of CTL-X.

Key words : temporal logic, synchronization, reachability graph, parallel programs, model checking, parameterized state graph.

1. Introduction

The study of parallel programs requires to specify and verify parallel behaviours and properties. Temporal logic offers, among others, these two possibilities. Simple temporal logic formulae express the usual properties of parallel programs [Lamport 83], [Emerson 88]. The properties of finite state programs are checked by efficient inspection of the reachability graph [Lichtenstein 85], [Clarke 86], [Emerson 89], [Vergauwen 93].

However parallel and distributed programs usually involve a finite but unknown number of processes communicating through a control structure. Thus the usual verification methods are no more applicable to obtain a complete verification. We need to verify properties on such programs independently of the number of communicating processes. Several authors have already proposed solutions to overcome this problem.

In [Clarke 87] and [Sistla 87], algorithms for temporal model checking concerning programs with many similar processes are introduced. In the first paper, they consider tree logic. The basic idea is to find a sufficient number of processes to have the full behaviour of the program. Then the usual verification methods are applied. However the search of the "sufficient number" is partially automatic. In the second paper, automatic verifications are obtained by restricting either the model or the formulae language. In [Wolper 89], an alternative approach is proposed where induction on formulae is used to verify that a given invariant remains true if the number of processes is increased. There is no way to find such an invariant and to decide if it exists. We must have the intuition of the useful invariant.

Our work is close to the one presented in [Clarke 87]. Our model is composed of two synchronized automata. One represents the behaviour of the processes and the other the behaviour of the control. The model can be instantiated by the number of processes. Then, it represents the program for the fixed number of processes. We give in [Vernier 93.a] an algorithm to directly build a parameterized state graph where each state represents a family of states of an ordinary computation. The construction procedure may fail; however in practice it usually terminates with success. Furthermore this graph is a high-level representation of the behaviour of the program. With this parameterized state graph, we immediately obtain the reachability set for any large enough number of processes. The algorithm presented in this paper is a parameterized temporal logic verification algorithm. We verify CTL-X formulae on the parameterized state graph. We have adapted the decision procedures presented in [Clarke 86] to take into account the specific features of the parameterized states.

The remainder of the paper is organized as follows. In section 2, we present the model we use for parameterized parallel computations. In section 3, we give a general survey of the parameterized state graph building algorithm. Section 4 presents an algorithm to evaluate temporal logic formulae on the parameterized state graph. We conclude in section 5 and we present the future extensions of our work.

2. A Model for Parameterized Parallel Computations

Many parallel algorithms proposed to ensure critical sections, distributed termination or to establish synchronization states are based on the (symmetrical) definition of a finite but indefinite number of processes communicating by some kind of control structure (shared variables, message passing, …). These algorithms are supposed to be correct whatever the number of processes. However one seldom find formal proofs of such algorithms. In this section we define a model that is suitable to lead to a formal proof independent of the number of processes.

The model we define is called PPCM (Parameterized Parallel Computations Model). It is composed of two automata: one representing the behaviour of the processes and the other the behaviour of the control structure. The control structure allows access to resources or communication between the processes. A PPCM is a family of parallel programs, one per cardinality of processes. Each automata is defined by a set of states, a set of transitions and two flow relations between states and transitions. Each transition has one source state and one target state. The backward flow relation associates to each transition its source state and the forward flow relation its target state.

Definition 2.1 *PPCM : a Parameterized Parallel Computations Model*

A parameterized parallel computations model P = $<G_p, G_c>$ is defined by:

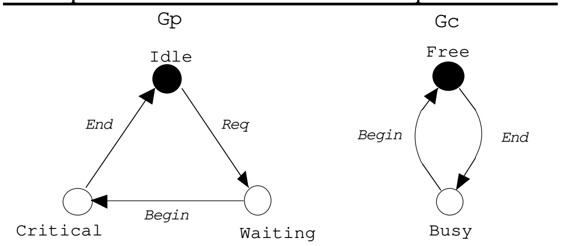
• The process automaton $G_p = <S_p, T_p, \Gamma_p^-, \Gamma_p^+, start_p>$ where :
  - S_p is the set of states of a process,
  - T_p is the set of transitions of a process,
  - Γ_p^- (resp. Γ_p^+) a function from T_p to S_p is the backward (resp. forward) flow relation between states,
  - $start_p \in S_p$ is the initial state of a process,

• The control automaton $G_c = <S_c, T_c, \Gamma_c^-, \Gamma_c^+, start_c>$ where :
  - S_c is the set of states of the control,
  - T_c is the set of transitions of the control,
  - Γ_c^- (resp. Γ_c^+) a function from T_c to S_c is the backward (resp. forward) flow relation between states of the control,
  - $start_c \in S_c$ is the initial state of the control.

• $S_p \cap S_c = \varnothing$ but $T_p \cap T_c$ is generally non empty and represents the synchronization transitions.

A PPCM does not enable direct synchronization between two processes. However this extension is straightforward.

Example 1: *A critical section between processes*



`The states are represented by the circles and the transitions by the arcs. The blackened states are the initial ones. Initially all the processes are idle. A process submits a request and waits until the control authorizes it to begin its critical section. At the end of the critical section, the control can accept new requests. {Begin, End} is the synchronization transitions set.`

We now state the definition and the semantics of an instantiated PPCM that is a program with a fixed number of processes. If *n* is the number of processes, the instantiated PPCM is composed of *n* copies of G_p and one copy of G_c. The synchronizations can only be executed between one copy of G_p and G_c.

Definition 2.2 *Instantiated PPCM*

Let P = $<G_p, G_c>$ be a PPCM and *n* be a positive integer (the number of processes), then P[n] = <S[n], R[n], start[n]>, the instantiated PPCM, is the parallel program defined by:

• S[n] is the set of possible states defined by: $S[n] = S_p^n \times S_c$.
An item s of S[n] is denoted by a tuple $s = <s_1, \dots, s_n, s_c>$. In the rest of the paper an item s is called an ordinary state since it is not parameterized.

• start[n] ∈ S[n] is the initial ordinary state defined by: $start[n]=<start_p,\dots, start_p, start_c>$

• R[n] is a relation between ordinary states of S[n] defined by:
  s R[n] s' where s, s' ∈ S[n] if and only if $\exists\, t \in T_p \cup T_c$ with
  (1)
  - $t \in T_p \Rightarrow \exists\, i \le n$ with $s_i = \Gamma_p^+(t)$ and $s_i' = \Gamma_p^+(t)$ and $\forall j \ne i, s_j = s_j'$
  - $t \notin T_p \Rightarrow \forall j \le n, s_j = s_j'$
  (2)
  - $t \in T_c \Rightarrow s_c = \Gamma_c^-(t)$ and $s_c' = \Gamma_c^+(t)$
  - $t \notin T_c \Rightarrow s_c = s_c'$.

Additional remarks

- S[n] defines the set of possible states but not the set of reachable states. The accessibility set is the transitive closure of the relation R[n] applied to $start_n$.

- R[n] defines the semantics of the program. A step of the program can be done by a process "i", the control or simultaneously both if the transition t belongs to T_p and T_c.

Example 2: *A computation of a parallel program*

```
Let P be the program of the example 1.
Then a possible computation of P[2] is
given by:
- From <Idle, Idle, Free> to <Waiting,
Idle, Free> using transition Req
activated by process 1
- From <Waiting, Idle, Free> to
<Waiting, Waiting, Free> using
transition Req activated by process 2
- From <Waiting, Waiting, Free> to
<Critical, Waiting, Busy> using
transition Begin activated by process 1
and the control
...
```

The aim of this paper is to check, when possible, properties of the programs P[n] for all values of n. At first we look at the reachability problem and this is the subject of the next section.

3. Parameterized State Graph

To be sure that a program, modelled by a PPCM, verifies some properties expressed as temporal logic formulae, we need first to build its reachability graph.

To study parameterized models, we have to build a parameterized reachability graph that represents all the reachable states and all the possible computations of the model. The graph must not represent particular cases of behaviour.

In many cases, the behaviour of the programs P[n] may be particular for some but a finite number of values of *n*. Mutual exclusion programs for *n* processes may have a special behaviour if there is only one process. We want to study the program when the value of *n* does not represent a special behavioural case. Usually, there is a minimum necessary number of processes to have no special behaviour of the program. This minimal bound is computed during the building of the parameterized state graph. As we do not represent special behaviours in the graph, we forbid the executions of transitions that imply a value assignment to *n*.

We briefly present here the features of the parameterized state graph. In [Vernier 93.a], the parameterized state graph building algorithm is described.

3.1 Parameterized States

Each parameterized state of the reachability graph represents a set of ordinary states for any P[n]. The key point of the representation is to associate an integer called a marking and a relational operator "=" or "≥" to each state of S_p. The intended meaning is the following:

- If "=x" is associated to a state, then exactly x processes are in this state.
- If "≥x" is associated to a state, then at least x processes are in this state.

These attributes are also used for the states of S_c. As there is only one control process we require that:

- only equality can be used,

- the marking can be either 0 or 1,

- there is exactly one state for which the marking is 1.

The used notation is close to the one presented in [Emerson 90]. In this paper, they are focusing on testing satisfiability rather than model checking.

The two attributes are defined by two functions associated with a parameterized state. If ps is a parameterized state then ps.OP and ps.M denote respectively the operator and the marking functions.

Notations

- We denote a parameterized state by $\{s\ OP(s)\ M(s)\}_{s \in S_p \cup S_c}$, forgetting all the states s for which OP(s) is "=" and M(s) is 0. Moreover the only marked state of S_c is represented without its relation and its marking (necessarily equal to "=" and 1).

- When necessary, we prefix M and OP by the parameterized state like for instance ps.OP(s).

Example 3: *Parameterized states for the critical section model*

```
The initial parameterized state of the
critical section model is given by:
{Idle≥1, Free} which denotes that the
control is in state Free and all the
processes are in state Idle.
The following parameterized state
denotes that one process is in its
critical section, two others are waiting
for the critical section and the rest of
the processes are in state idle:
{Idle≥1, Waiting=2, Critical=1, Busy}.
```

The relation between ordinary and parameterized states is intuitive. An ordinary state belongs to a parameterized one if the distribution of processes among states fulfils the relations expressed by the parameterized state. A parameterized state ps is included in an other one, ps', if and only if the set of ordinary states represented by ps is included in the set of ordinary states represented by ps'. This is noted ps $\subseteq$ ps'. If ps is not equal to ps', then ps $\subset$ ps'.

Example 4: *Correspondence between states*

```
Let ps be the following  parameterized
state:
{Idle≥1,Waiting=2,Critical=1,Busy} then
in P[5],
os = <Idle, Waiting, Waiting, Critical,
Idle, Busy> belongs to ps.
```

In the previous example if we consider P[3], ps represents none of its ordinary states. This correspondence between states shows that a parameterized state, ps, represents a set of ordinary states of P[n], {os}, only if *n* is greater than a bound defined by M. This bound is equal to $\sum_{s \in Sp} M(s)$.

3.2 Parameterized Transitions

To build the parameterized reachability graph, we need to define parameterized transitions between parameterized states. This definition must be consistent with the definition of a transition between ordinary states. A transitions can be executed from a parameterized state ps if and only if it can be executed from each ordinary state represented by ps. For each ordinary state represented by ps, its successor state, after the execution of a transition t, is represented by one successor parameterized state of ps, after the execution of the same transition. If ps' is a successor of ps by transition t, then for each ordinary states, os', represented by ps', there is an ordinary state, os, represented by ps such that os' is the successor of os by transition t. Roughly speaking, a parameterized transition from a parameterized source state to a parameterized target state is possible if:

- there is enough marking in the source state.

- the target state is obtained by consistent decreases and increases.

Let ps and ps' be two parameterized states. If we want to execute the transition Req from parameterized state ps, there must be at least one process in state Idle, e.g., ps.M(Idle) $\geq$ 1. If ps' is the reached parameterized state when the transition Req is executed then ps'.M(Idle) = ps.M(Idle)-1 and ps'.M(Waiting) = ps.M(Waiting)+1 since by transition Req one process moves from state Idle to state Waiting.

However when a state s fulfilling s $\geq$ 1 is decreased, two cases may happen:

- only one process was in state s, then in the target state one have s = 0. As the number of processes is parameterized, this is possible only if another state s' fulfils s'$\geq$j. Otherwise it would assign a value to the parameter *n*.

- at least two processes were in state s, then in the target state one have s $\geq$ 1.

The parameterized transitions include these two cases that can simultaneously happen.

Example 5: *Transitions between states*

```
Let ps be the following parameterized
state:
{Idle≥1, Waiting≥2, Free}
Then ps leads to ps1 by transition begin
where:
ps1 = {Idle≥1, Waiting≥1, Critical=1,
Busy}.
Also  ps  leads  to  ps2  and  ps3  by
transition Req where:
ps2 = {Idle≥1, Waiting≥3, Free} and ps3 =
{Waiting≥3, Free}.
```

3.3 Building of the Parameterized State Graph

Since we are given an initial parameterized state and a transition rule, it should be easy to develop an accessibility tree (or graph). However such a structure would be generally boundless. Thus we must take some decisions during the building of the tree. The procedure avoiding the infinite development is similar to the one of Karp and Miller [Karp 69] [Finkel 90] and we just give here an informal presentation.

Let ps and ps' be two parameterized states, ps $\leq$ ps' if for each place p:

- ps.M(p) $\leq$ ps'.M(p) and

- ps.OP(p) $\leq$ ps'.OP(p) (we state that "=" < "$\geq$").

As soon as we find two states ps and ps' on the same branch of the reachability tree such that ps $\leq$ ps', three cases may happen:

(a) ps = ps', so we stop the development of this branch.

(b) ps $\neq$ ps' and the sequence of transitions that leads from ps to ps' is identifiable as "a state change from s to s' for one process", then we transform ps' in ps except for s' where ps.OP(s') is set to "$\geq$". Moreover if ps.M(s')=0 then ps'.M(s)=1.

(c) otherwise we stop the whole building : this is the failure case of the algorithm.

A sequence of transitions is "a state change from s to s' for one process" if after its execution:

- one process has moved from s to s',
- all the other processes and the control are in the same state.

The change of operator "=" to "$\geq$" is justified by the fact that this sequence can be iterated as long as there are processes in state s. After each execution of the sequence, a process has moved from the state s to s'.

In practice, case(c) is rarely encountered. It is mainly met when the execution of sequences is dependent on some properties of the parameter, e.g., if the behaviour of the algorithm is different when the number of processes is odd or even. In such cases, we can not represent in a single graph the behaviour of the algorithms whatever the number of processes.

Let us show how the building works on one branch.

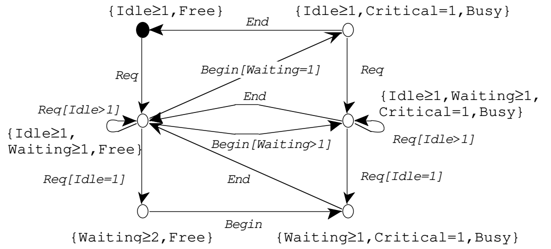
Example 6: *A branch of the accessibility tree*

We start from the initial parameterized marking
pstart = {Idle≥1, Free}
We can use transition Req and obtain
$pstemp_1$ = {Idle≥1, Waiting=1, Free }
$pstemp_1$ is greater than pstart and we can identify Req to a process change from Idle to Waiting so we change $pstemp_1$ to:
ps_1 = {Idle≥1, Waiting≥1, Free}
We can use the transition Req and obtain
$pstemp_2$ = {Idle≥1, Waiting≥2, Free}
Again we transform $pstemp_2$ but this time $ps_2 = ps_1$ so we stop the building of this branch.

We do not compute the successor states of a state ps if there exists a parameterized state ps', already computed, such that ps is included in ps'. While computing the successor states of ps' we also compute those of ps.

The termination of our algorithm is now ensured by the order properties of $\mathbb{N}^k$. So if we get success we identify the nodes ps with ps' if ps ⊂ ps'. Therefore, we obtain a parameterized state graph.

Example 7: *The parameterized graph of the critical section model*

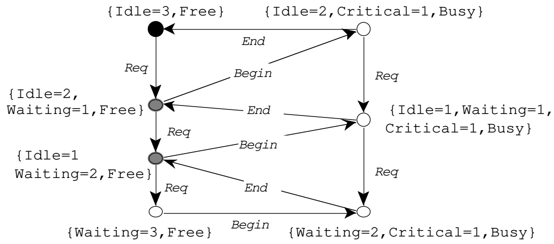


This parameterized state graph represents the reachability states of P[n] with n≥3.

Let P be a PPCM, let G be the parameterized graph of P (if the building was successful). G represents the behaviour of P[n] if and only if n is greater than n_0 where n_0 is the maximum of $\sum$ps.M(s) for ps ∈ G and s ∈ S_p. Therefore, for each $n \geq n_0$, G can be unfolded to G[n], the state graph of P[n].

Example 8: *Unfolding of the parameterized state graph for n=3: G[3]*

The gray states are represented by the same parameterized state {Idle≥1, Waiting≥1, Free} in the parameterized state graph of the example 7.



For each execution sequence of G[n], $n \geq n_0$, there is an equivalent possible sequence in G. For each possible sequence of G, there is an equivalent sequence in G[n], $\forall n \geq n_0$. Therefore, the result of the verification of properties on G is valid for each G[n], $n \geq n_0$. The proof is given in [Vernier 94].

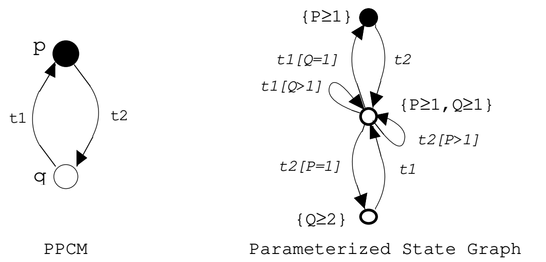
3.4 Detection of "real" circuits

An important feature of the CTL formulae evaluation algorithms is the detection of "real" circuits. They allow to find executions that may infinitely be repeated. By "real" circuits we mean circuits that appear in the graphs G[n]. While the computation of the parameterized state graph, the built circuits are not always "real".

On the graph of the example 7, all the "real" circuits are already computed.

Example 9: *Parameterized state graph without "real" circuits*

The following PPCM represents the behaviour of processes that can be in two states and move from one to the other without constraints.

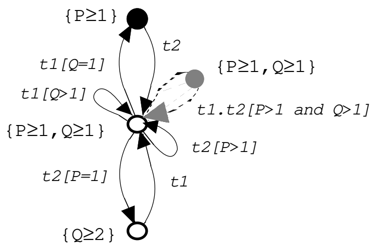


The two circuits of the graph are t1 and t2 from the parameterized state {P≥1,Q≥1} to itself. They can be executed only when P>1 and Q>1. A real circuit can be composed of these two transitions and it corresponds to the execution of the transition t1 (resp. t2) a λ_1 (λ_2) number of times. The system of equations we have to solve represents the change of the number of processes in the places P and Q when the supposed circuit is executed. If there is a strictly positive integer solution then there is a real circuit that leads from {P≥1,Q≥1} to itself. The system of equations is:

state P: $\lambda_1 - \lambda_2 = 0$

state Q: $\lambda_2 - \lambda_1 = 0$

This equation has a solution. Therefore, there is a "real" circuit, holding t1, that leads from {P≥1,Q≥1} to itself. This circuit can be executed only if P>1 and Q>1. The parameterized state graph with real circuits is as follows :



Parameterized State Graph with "real" circuits

The set of the real circuits can be computed from the initial circuits by solving positive integer equations [Ginsburg 64].

4. Parameterized Evaluation of CTL-X Formulae

Everyone that models a system has some ideas of the supposed properties of the system. To express these properties, temporal logic formulae are an appropriate language [Lamport 83], [Emerson 88]. We have chosen CTL-X.

CTL-X is a temporal logic reasoning about states and their successors. The existing algorithms are depth-first search algorithms [Clarke 86], [Vergauwen 93]. They are based on the following principle: they do not iterate the evaluation procedures on nodes of the graph that have already been visited. It is useful to visit one time a node to know if it verifies a formula. This last property is not respected for the nodes of the parameterized state graph. Each parameterized state defines a set of ordinary one. Therefore, the examination of new successors of an already visited parameterized state may allow to detect new ordinary states that verify the formula. To solve this problem, the algorithm attaches to each node a condition that defines the subset of ordinary states that fulfil the formula. When visiting a node, the subset is kept up to date. If it is stable, we do not continue the examination from this node.

4.1 CTL-X

We have chosen a branching time logic since it is more adapted as a linear to the structure of the state graph and to the properties we want to verify [Pnueli 85], [Emerson 86].

Although CTL-X has a more restrictive power than other languages (e.g., CTL*), we have chosen it for the following reasons:

• The verification of a CTL-X formula on a state graph has a complexity linear in the size of both the formula and the graph.

• Although CTL-X is a simple language, it is sufficient to express the usual properties such as safeness (i.e. a property f must be verified at each reachable state which is written (∀G f) in CTL-X) and liveness (i.e. from each state which verify g we can always reach a state that verify h which is written (∀G(g ⇒∀F h)) in CTL-X).

The version of CTL we use is named CTL-X and a formula is equal to:

- one of the two constants: True or False,
- a basic formula,
- a formula obtained by connectors And, Or, Not applied to formulae of CTL-X,
- a formula obtained by connectors ∀G, ∀F, ∀U, ∃G, ∃F, ∃U applied to formulae of CTL-X.

The basic formulae define the set of properties that can be verified at a parameterized state without inspection of its successors. Therefore they have the following syntax and semantics:

- {s ≥ 1}, means that at least one process is in state s,
- {s = n}, means that all the identical processes are in state s,
- {s op j}, j ∈ $\mathbb{N}$ and op {=, >} means that the number of processes in state s respects the relation "op j".

A parameterized state ps verifies:

- {s ≥ 1} only if ps.M(s) ≥ 1,
- {s = n} only if ps.OP(s) = "≥" and ∀s'≠s, ps.M(s')=0,
- {s op j} only if
  ps.OP(s) = "=" and ps.M(s) op j or
  ps.OP(p) ="≥" and j+∑ps.M(p') ≤ n_0, this condition ensures that the formula is verified whatever the possible value of the parameter is.

The verification, at a parameterized state ps, of a formula with temporal operator depends on its successors. The next paragraph states the semantics of the temporal operators.

- ∀ means for any path starting from the current state…
- ∃ means there is some path starting from the current state…
- Gp means …such that p is true for all states of the path
- Fp means …such that p is true for some state of the path
- pUq means …such that p is true on the path until q becomes true and q will eventually becomes true

The operators G and F are abbreviations of formulae written with the U operator.

We do not consider the X (successor state) operator since it allows to count the number of processes. Let F be the formula $\forall G(\text{Idle}\geq 1 \Rightarrow \exists X(\text{Idle}=0))$. F is verified if each time there is a process in state Idle, one of the successor states is such that there is no Idle process. Considering our example, F can be verified only if there is one process. Such formulae may represent special behaviours of the programs.

Example 10: *Some simple formulae about the critical section model*

```
Two processes are never simultaneously
in critical section:
∀G ({Critical ≤ 1})
Requests imply services:
∀G({Waiting ≥ 1} ⇒∀F{Critical ≥ 1})
```

4.2 Conditions on parameterized states

It may possibly be that two ordinary states, held by the same parameterized state, do not verify the same formulae. This case may happen only if from one ordinary state it is possible to empty a state while from the others it is not possible. Consider the parameterized state {Idle≥1, Waiting≥1, Free}, the ordinary states such that Idle>1 may not verify the same formulae as those where Idle=1. The condition, Cond(ps,f), defines the subset of ordinary states, included in ps, that verify f. The subsets can not be differentiated by the marking of the control states. The conditions concern only the states of the processes automaton. A condition is:

- an elementary condition Cond(ps,f) = Ce where
    - Ce = All (all the ordinary states included in ps verify f) or
    - Ce = None (no ordinary state included in ps verify f) or
    - Ce is a pseudo parameterized state where the operator associated to each place can be "=", ">" or "≥". In a condition, the states that do not appear are not constraint. The number of processes in these states depends only of the parameterized state with which the condition is associated. If ps={Idle≥1, Waiting≥1, Free} and Ce(ps,f)={Idle>1} then the ordinary states, included in ps, with more than one process in state Idle verify f. We can extend Ce(ps,f) to {Idle>1, Waiting≥1, Free}. A condition Ce(ps,f) such that $\sum \text{Ce(ps,f).M(p)} > n_0$ is not compatible with the parameterized state graph. Otherwise, Ce is not equal to None does not mean that for each parameter value, there is at least one ordinary state represented by ps that verifies the formula f. We want to test if a distribution of processes verifies a formula whatever the value of the parameter is.
- an union of elementary conditions Cond(ps,f) = $\cup \text{Ce}_i$ (if $\cup \text{Ce}_i$=ps then Cond(ps,f)=All).

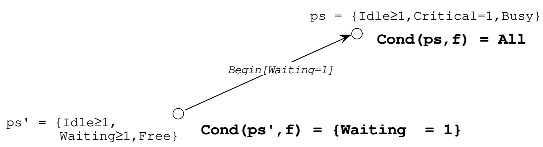
For the verification algorithm, we need to know how to compute the set of ordinary states included in ps that allow to reach the ordinary states of ps' that verify a condition, Cond, when $ps \xrightarrow{t} ps'$. This set, Trans(Cond,ps,t,ps'), is a condition defined as follows and is only computed when Cond ≠ None:

- let s and s' be the two states of S_p such that $\Gamma_p^-(t)$=s and $\Gamma_p^+(t)$=s',
- Cond = All then Trans(Cond,ps,t,ps') = All
- Cond = $\cup \text{Ce}_i$ and Trans(Cond,ps,t,ps') =$\cup$ (ps $\cap$ C'e_i) where:
    - $\forall \text{Ce}_i$
        if s appears in Ce_i then C'e_i.M(s) = Ce_i.M(s)+1, C'e_i.OP(s) = Ce_i.OP(s),
        if s' appears in Ce_i then C'e_i.M(s) = Ce_i.M(s)-1, C'e_i.OP(s') = Ce_i.OP(s'),
        $\forall$ s"$\notin$ {s,s'}, C'e_i.M(s") = Ce_i.M(s"), C'e_i.OP(s") = Ce_i.OP(s"),
- if t empties s then Trans(Cond,ps,t,ps') = Trans(Cond,ps,t,ps') $\cap$ {s=1},
- if t does not empty s then Trans(Cond,ps,t,ps') = Trans(Cond,ps,t,ps') $\cap$ {s>1}.

The set Trans^{-1}(Cond,ps,t,ps') defines the ordinary states of ps' that are reached by ordinary states of ps that verify Cond when $ps \xrightarrow{t} ps'$. We just notice that Trans(Trans^{-1}(Cond,ps,t,ps'),ps,t,ps')=Cond.

Example 11

```
We consider the following restriction of
the parameterized state graph of the
example 7 and the formula
   f = ∃(True U {Waiting=0}).
```




```
The parameterized state ps verifies the
basic formula {Waiting=0} therefore
Cond(ps,f)=All. When we restrict to this
part of the graph Cond(ps',f) =
Trans(All,ps',Begin[Waiting=1],ps) ∩ All
= {Waiting=1} ∩ All = {Waiting=1}. All
the ordinary states of ps' such that one
process is in state Waiting verify f.
```

4.3 Algorithm

The algorithm to evaluate CTL-X formulae on a parameterized state graph is similar to the one presented in [Clarke 86]. We have defined new procedures to evaluate formulae with temporal operators as $\forall[f_1 \text{ U } f_2]$ and $\exists[f_1 \text{ U } f_2]$. For place constraints, we only present the procedure to evaluate formulae as $\forall[f_1 \text{ U } f_2]$ while f_1 and f_2

have already been evaluated.

With each parameterized state and formula is associated a condition that defines the subset of ordinary states that verify the formula. We use an array of conditions (Marked) that associates with each parameterized state the set of ordinary markings that have already been visited. We also use a stack (ST) to store the parameterized states visited along a path and the path. When we push a parameterized state, we also push the transition that leads to it (if there is one). This way it is possible to know if some parameterized states are linked by a circuit ("real" or "false"). The function RealCircuit(ST,ps) returns the value true if the circuit between ps and ps is or hides a "real" one. The function Stacked(ST,ps) returns the value true if ps is in the stack ST.

```
procedure AU(f)
{f = ∀[f1 U f2]}
begin
    for all ps ∈ G do
        begin
            Marked(ps) := Cond(ps,f2);
(a)         Cond(ps,f) := Cond(ps,f1)∪Cond(ps,f2)
            Modif(ps) := false;
            Liste(ps) := Ø;
        end
    for all ps ∈ G do
       if Marked(ps)<>All then VerifAU(f,ps,All)
end
```

The procedure VerifAU(f,ps,ens) checks the formula f at ordinary states of ps that verify the condition ens. While the execution of this procedure, the ordinary states included in ps that verify f_1 and not f_2 are deleted from Cond(ps,f). When the procedure ends, then Cond(ps,f) ∩ ens gives the ordinary states of ens that verify f. Modif(ps) indicates if the condition associate with ps has to be modified. A modification is due to the detection of a state change sequence, from p' to p", that leads from ps to ps. The set Liste(ps) holds the places p' of the state change sequences that have been detected.

```
VerifAU(f,ps,ens)
begin
    if (ens ⊆ Marked(ps)) and not(Stacked(ST,ps))
        then return;                              {1}
    if Stacked(ST,ps) then
        if RealCircuit(ST,ps) then                {2}
            begin
(b)             Cond(ps,f) := Cond(ps,f) - ens;
                return
            end
        else                                      {3}
            { there is a state change sequence from
            p' to p" }
            if (∑ens.M(p) > n0) then
                begin
                    Modif(ps):=true
                    Liste(ps):=Liste(ps)∪{p'}
                end
            return;
```

{In case (1), the ordinary states have already been studied. In case (2), ps is linked to itself by a "real" circuit, therefore there is a loop of ordinary states that never fulfil f_2. In case (3), the circuit is a "false" one, we can not say, at this step, if the states defined by ens verify f or not. We only know that the executed sequence can be done again and then the conditions associated to the place p' have to be modified}.

```
    Marked(ps) := Marked(ps) ∪ ens;
    ens2 := Cond(ps,f1) ∩ ens;
    {set of states of ens that may verify f}

    Push(ST,ps);
    ens3 := Ø;
    for all ps' such that ps -t-> ps' do
        begin
            ens4 := Trans⁻¹(ens2,ps,t,ps');
            {set of states of ps' that are reached
            by states of ens2 with the transition t}

            ens3 := ens3 ∪ ens4;
            {set of states reached by states of ens2}

            VerifAU(f,ps',ens4);
(c)         Cond(ps,f) := Cond(ps,f)-
                Trans(ens4- Cond(ps',f),ps,t,ps');
            {we remove from Cond(ps,f) the states
            of ens2 that lead to states of ens4 that
            do not verify f}
            if Cond(ps,f)=None then
                begin
                    Pop(ST);
                    return
                end
        end;

        Cond(ps,f) := (Cond(ps,f) -
        (ens2-Trans(ens3,ps,t,ps')))∪Cond(ps,f2);
        {we remove from Cond(ps,f) the states of
        ens2 that do not verify f2 and have no
        successor}

        if Modif(ps) = true then
            begin
                { Cond(ps) = ∪Cei }
                for each p in Liste(ps)
                    For each Cei, Cei.OP(p) = "≥"
            end
    Pop(ST);
end;
```

To define the set of ordinary states of ps that verify $\exists[f_1 \text{ U } f_2]$ we study all the successors of ps and add to Cond(ps,f) the states that verify f_1 and that lead to one that verify $\exists[f_1 \text{ U } f_2]$. The algorithm is, roughly speaking, the same as for $\forall[f_1 \text{ U } f_2]$ apart from the lines a,b and c.

(a') Cond(ps,f) := Cond(ps,f_2);
(b'), (d') we remove these lines;
(c') Cond(ps,f) := Cond(ps,f) $\cup$
Trans(Cond(ps',f),ps,t,ps');

Property: The conditions Cond(ps,f) = $\cup Ce_i$ are such that $\sum Ce_i.M(s) \leq n_0$. Otherwise, the condition Ce_i does not appear in Cond. It is not compatible with the parameterized state graph and the bound n_0.

This property ensures that the verification algorithm ends.

5. Conclusion

In this paper, we have introduced a model to reason about parallel algorithms including a finite but unknown number of processes. Then we have presented a procedure to obtain a parameterized representation of the reachability set. This procedure fails when we are not able to represent, with a single graph, the behaviour of the model whatever the value of the parameter is. The success of this procedure often happens in real cases. We have developed a procedure to evaluate the formulae of the language CTL-X on the parameterized state graph.

Compared with the work presented in [Clarke 87], we also define a sufficient number to have the full behaviour of the parameterized program. Clarke and Grümberg do not find this bound automatically. They do not specify how to determine if such a bound exists. In our work we find a bound automatically during the building of the parameterized state graph. Another interest of our work is that we do not compute the reachable states only for the program with the sufficient number of processes. We compute the reachable states for programs with any number of processes greater or equal to the bound.

Further works will deal with the possible extensions of the model. The model presented in [Vernier 93.b] allows to distinguish the behaviour of one process. This distinction offers two possibilities: to prove fairness properties (e.g., if a process wants to access critical section it will success) and to model problems where one process has a specific behaviour. This distinguished process can be dynamically changed. The parameterized state graph building algorithm and the evaluation algorithm work on this model. A lack of this model is the inability to consider order on processes. It will allow to model algorithms that assume that processes are on a ring and can communicate only with their neighbours. The difficulty lies in the building of the parameterized state graph. We have to be able to keep links between processes.

Another improvement of our work is the consideration of more complex formulae languages such as CTL*, FCTL [Emerson 85]. The model checkers that consider such logics have to be able to compute strongly connected components of the state graph. Therefore we will be able to deal with these logics if we are able to detect strongly connected components on the parameterized state graph.

References

[Clarke 86] 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, April 1986, pages 244-263.

[Clarke 87] Clarke E.M., Grümberg O., *Avoiding The State Explosion Problem in Temporal Logic Model Checking Algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, August 1987, pages 244-303.

[Emerson 85] Emerson E.A., Lei C., *Modalities for model checking: Branching time strikes back*, Proc. 12th Annual Symposium on Principles of Programming Languages, New-Orleans, Louisiana, January 1985, pages 84-96.

[Emerson 86] Emerson E.A., Halpern J.Y., *'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time*, Journal of ACM, vol. 33, n°1, 1986, pages 151-178.

[Emerson 88] Emerson E.A., Srinivasan J., *Branching Time Temporal Logic*, Branching Time and Partial Order in Logics and Models for Concurrency, Workshop on Linear Time, LNCS 354, 1988, pages 123-172.

[Emerson 89] Emerson E.A., Sadler T., Srinivasan J., *Efficient Temporal Reasoning*, Proc. of the 16th ACM Symposium of Principles of Programming Languages, Austin, Texas, 1989, pages 166-178.

[Emerson 90] Emerson E.A., Sadler T., *A Decidable Temporal Logic to Reason about Many Processes*, Proc. 9th ACM Symposium on Principles of Distributed Computing, 1990, pages 233-245.

[Finkel 90] Finkel A., *A minimal Coverability Graph for Petri Nets*, Proc. 11th International Conference on Application and Theory of Petri Nets, Paris, France, June 1990, pages 233-245.

[Ginsburg 64] Ginsburg S., Spanier E., *Bounded Algol-like languages*, Trans. Amer. Math. Soc. 113, 1964, pages 333-368.

[Karp 69] Karp R.M., Miller R.E., *Parallel Program Schemata*, Journal of Computer science, vol. 3, n°2, May, 1969.

[Lamport 83] Lamport L., *What good is temporal logic?*, Proc. IFIP 9th World Computer Congress, Paris, France, 1983, Information processing 83, R.E.A. Mason editor, North-Holland, pages 657-668.

[Lichtenstein 85] Lichtenstein O., Pnueli A., *Checking That Finite State Concurrent programs*

Satisfy Their Linear Specification, Proc. 12th ACM Symposium of Principles of Programming Languages, New-Orleans, Louisiana, January 1985, pages 97-107.

[Pnueli 85] Pnueli A., *Linear and branching structures in the semantics and logics of reactive systems*, Proc. 12th International Colloquium on Automata, Languages and Programming, Springer Verlag, 1985, pages 15-32.

[Sistla 87] Sistla A.P., German S.M., *Reasoning with Many Processes*, Proc. Symposium on Logic in Computer Science, Ithaca, New-York, June 1987, pages 138-152.

[Vergauwen 93] Vergauwen B., Lewi J., *A Linear Model Checking Algoritm for CTL*, Proc. 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 1993, LNCS 715, Springer Verlag, pages 447-461.

[Vernier 93.a] Vernier I., *Parameterized Reachability Graph for Parametrized Petri Nets*, Internal report, 1993.

[Vernier 93.b] Vernier I., *Specification And Verification of Parametrized Parallel Programs*, Proc. 8th International Symposium on Computer and Information Sciences, Istanbul, Turkey, November 1993, pages 622-625.

[Vernier 94] Vernier I., Evaluation Paramétrée de Formules de Logique Temporelle, Internal report, IBP, Laboratoire MASI, to appear.

[Wolper 89] Wolper P., Lovinfosse V., *Verifying Properties of Large Sets of Processes with Network Invariants*, Proc. International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France 1989, LNCS 407, pages 68-80.