On-line model-checking for finite linear temporal logic specifications

Claude Jard, Thierry Jéron

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1041

Programme 3

ON-LINE MODEL-CHECKING FOR FINITE LINEAR TEMPORAL LOGIC SPECIFICATIONS

14p.

Claude JARD
Thierry JERON

Mai 1989

RR-1041

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication interne n° **469**, Mai 1989, 16 pages

On-Line Model-Checking for Finite Linear Temporal Logic Specifications

Claude JARD, Thierry JERON

e-mail: jard@irisa.fr, jeron@irisa.fr

Abstract

Model-checking is the basis of several verification tools. It allows to check if a finite state program satisfies a set of temporal logic formulas. The main limitation is the size of the memory needed to record the state graph. Avoiding state space explosion is necessary to improve the applicability of such verification tools. For that aim, we explore an approach called "on-line model-checking" where satisfiability is checked during the state generation process. We deal with the simple context of checking finite computations against linear temporal logic properties. The logic specification is first translated into a finite automaton. This one values the system states during enumeration. Validity can be decided in finite time provided a memory larger than the state graph diameter. Precise algorithms and some results of experiments are given.

Evaluation au fil de l'eau de formules de logique temporelle linéaire sur des graphes finis

Résumé

L'évaluation de formules de logique sur des graphes d'états finis est la base de différents outils de vérification. Cette technique permet de vérifier qu'un programme ayant un nombre fini d'états satisfait un ensemble de formules de logique temporelle. La limitation principale est la taille de la mémoire nécessaire pour conserver le graphe d'états. Pour améliorer l'applicabilité de tels outils de vérification, il est nécessaire de limiter l'explosion de l'espace d'états. Dans ce but nous explorons une approche appelée "vérification au fil de l'eau" où la satisfaction est vérifiée au cours du processus de génération des états. Nous nous plaçons dans le contexte simple de la vérification de propriétés exprimées en logique temporelle linéaire sur des calculs finis. La spécification logique est d'abord compilée en un automate fini. Celui-ci value les états du système pendant l'énumération. La validité est décidable en temps fini pourvu que l'on dispose d'une taille mémoire supérieure au diamètre du graphe. Des algorithmes précis sont fournis ainsi que des résultats expérimentaux.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Contents

1 Introduction

If we restrict our attention to finite state programs, i.e. programs in which the variables (including the communication channels if any) range over finite domains, then the whole program can be represented as a (generally large) finite graph consisting of states and transitions connecting them.

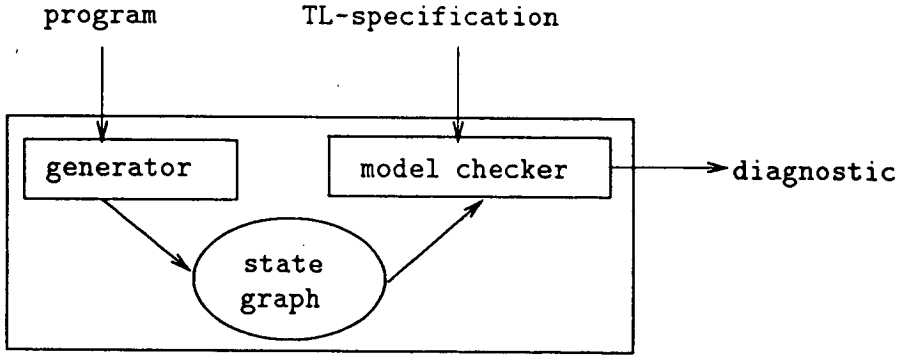
Each transition of this state graph is valued with the atomic action which has just changed the state. Consequently, a finite state program can be viewed as a finite model over which temporal formulas can be evaluated. Checking that a given finite model satisfies a given temporal formula is what one calls "model-checking".

We consider the linear time version of temporal logic and atomic propositions as actions [Tho81]. This latter point slightly differs from the classical more general form where, instead of the set of actions, the finite set of states S is considered and a map from the set of propositional variables into 2^S is given, stipulating for any variable p those states where p is assumed true. Our terminology is no essential restriction and simplifies the transition to the automata framework we use thereafter.

Models for linear logic are totally ordered computations. We restrict our attention to finite computations. Extension to the infinite case will be discussed. Hence, we shall say that a finite state graph satisfies a linear temporal formula if all the finite computations defined by this graph (often called the language associated to a transition system) satisfy the formula.

Classical model-checking as implemented in EMC [CES83] or XESAR [RRSV87] (for a branching time temporal logic) is illustrated in figure 1. It is assumed that the complete state graph is available before entering checking. This allows to use efficient fixpoint algorithms to evaluate formulas. The main limitation is the amount of memory needed to record the state graph (the best tool we know in that matter is XESAR which may accept a few hundred thousand of states on 8 MBytes SUN Workstations). Because of the necessity, for the graph construction, to compare each new state with those already generated, the performance collapse is unavoidable, whatever coding and access techniques may be used. Avoiding the state explosion problem was discussed in [CG87] for a branching time logic. The solution was to partition the essentially identical processes into equivalence classes.

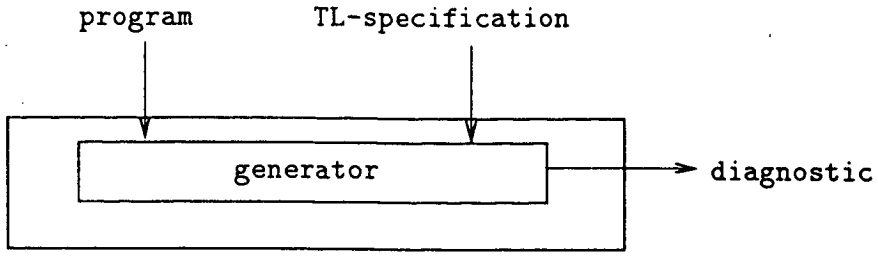
Figure 1: a verifier



Our paper presents a first step to a complementary approach which tends to considerably decrease the state space needed to perform checking. *The basic idea is to check during the state enumeration.* For that aim, the temporal logic specification must be executed [Sif86]. It is done by first translating the logic specification into a finite automaton. Here is the main algorithmical difficulty. The automaton will value the system states during enumeration. We will show that decision of validity or rejection can be reached in finite time providing a large enough memory to

store a number of states equal to the depth of the state graph. This way of working, that we call "on-line model-checking" is illustrated in figure 2. A related technique is formally described in [LP84] where, to each system state is associated a set of formulas calculated from the initial linear temporal logic formula.

Figure 2: an "on-line" verifier



The exposition is structured as follows:

We first present the considered temporal logic. In the second part, we recall an effective algorithm we designed to translate logic formulas into finite automata when they are interpreted over finite computations. Section 3 exposes the model-checking algorithm and its properties. We then discuss an extension to the infinite case for a particular (but rather large) class of formulas, called "deterministic" which exactly corresponds to the languages accepted by deterministic Büchi automata. We shall conclude by a discussion and a few prospects.

2 The considered TL

Let Σ be a finite alphabet (the set of observable events, augmented with a special symbol τ denoting the invisible actions). The temporal formulas over Σ ($\mathcal{F}$) are built up from the atomic propositions a (for $a \in \Sigma$) using the boolean connectives $\neg$, $\wedge$, the unary temporal operator $\bigcirc$ ("next"), the binary temporal operator $\mathcal{U}$ ("until"), and brackets [Pnu86].

Some abbreviations are also considered : $\vee$ (or), $\supset$ (implies) and $\equiv$ are defined in the usual way, $\forall f \in \mathcal{F}, \bigodot f \equiv \neg \bigcirc \neg f$ (strong next), $\Diamond f \equiv \top \mathcal{U} f$ (eventually) and $\Box f \equiv \neg\Diamond\neg f$ (always).

Examples of formulas over $\Sigma = \{a, b, \tau\}$ are

$$\varphi_1 = \Box(a \Rightarrow \bigodot(\neg a \mathcal{U} b))$$
$$\varphi_2 = \Diamond a \ \mathcal{U} \ \Box b$$

The logical formulas are interpreted over finite nonempty sequences $\sigma = \sigma_0\sigma_1...\sigma_n$ ($n \geq 1$). The length of σ is denoted by $\mid \sigma \mid$.

The satisfaction relation $\models$ between pairs (σ, i) (where $0 \leq i <\mid \sigma \mid$) and formulas φ is inductively defined as follows :

$$\forall a \in \Sigma, \forall \varphi, \psi \in \mathcal{F},$$

$$\begin{array}{lll}
(\sigma,i) \models \top & & always \\
(\sigma,i) \models a & iff & \sigma_i = a \\
(\sigma,i) \models \neg\varphi & iff & not\,((\sigma,i) \models \varphi) \\
(\sigma,i) \models \varphi \wedge \psi & iff & ((\sigma,i) \models \varphi)\ and\ ((\sigma,i) \models \psi) \\
(\sigma,i) \models \bigcirc\varphi & iff & (i =\mid \sigma \mid -1)\ or\ ((\sigma,i+1) \models \varphi) \\
(\sigma,i) \models \varphi\mathcal{U}\psi & iff & \exists j,\ i \leq j <\mid \sigma \mid, \\
 & & ((\sigma,j) \models \psi)\ and\ (\forall k,\ i \leq k < j,\ (\sigma,k) \models \varphi)
\end{array}$$

We say that a computation σ satisfies a formula φ ($\sigma \models \varphi$) if and only if $(\sigma, 0) \models \varphi$. We can easily extend the possible interpretations with the empty sequence λ by considering $\lambda \models \varphi$ as defined by $(\lambda, -1) \models \varphi$.

A temporal formula φ over an alphabet Σ defines a set $L(\varphi, \Sigma)$ of finite sequences such that :

$$L(\varphi, \Sigma) = \{\sigma \in \Sigma^* \mid \sigma \models \varphi\}$$

For example, considering the formulas φ_1 and φ_2 given above,

$$L(\varphi_1, \Sigma) = (b \cup \tau)^* \cup (a\tau^* b(b \cup \tau)^*)^*$$

(set of finite words such that between two a there is at least a b)

$$L(\varphi_2, \Sigma) = b^+ \cup \Sigma^* a b^+$$

(while the suffixe does not only contain some b, there is a a in future)

Let us now consider a finite transition system $\mathcal{S} =< Q, E, \delta, q_0 >$ where Q is the finite set of states, E the alphabet of actions, δ the transition relation and q_0 the initial state as usual.

The associated finitary language over an alphabet Σ is the set of all the possible computations in which the invisible actions are renamed by τ.

$$L(\mathcal{S}, \Sigma) = \{\phi(\sigma), \sigma \in E^* \mid \delta(q_0, \sigma) \in Q\}$$

where ϕ is the renaming function which transforms the elements of $E - \Sigma$ into τ.

We will say that a transition system satisfies a formula φ if and only if all its finite computations on Σ satisfy φ.

$$\mathcal{S} \models \varphi \ \ iff \ \ \forall \sigma \in L(\mathcal{S}, \Sigma), \ \ \sigma \models \varphi$$

That is to say :

$$\mathcal{S} \models \varphi \ \ iff \ \ L(\mathcal{S}, \Sigma) \subseteq L(\varphi, \Sigma)$$

3 From TL to finite automata

The theory of linear time logic was linked to the automata theory several years ago. We know since 1968 that a language is TL-definable if and only if it is first-order definable. And, since the early seventies, we know [MNP71] that first-order definability can be characterized elegantly in terms of "star-free" languages (a star-free language is a regular language included in the closure of the finite word-sets under concatenation and boolean operations only).

As an illustration, here is the star-free representations of $L(\varphi_1, \Sigma)$ and $L(\varphi_2, \Sigma)$ considered above as examples :

$$L(\varphi_1, \Sigma) = \neg[\Sigma^* a \neg(\Sigma^* b \Sigma^*) \cup \Sigma^* a \neg(\Sigma^* b \Sigma^*) a \Sigma^*]$$

$$L(\varphi_2, \Sigma) = b\neg(\Sigma^*(a \cup \tau)\Sigma^*) \cup \Sigma^* a b \neg(\Sigma^*(a \cup \tau)\Sigma^*)$$

where Σ^* is an abbreviation for $\neg\emptyset$.

This proposition was first applied in [MW84] to show the possibility of synthesizing synchronization skeletons of protocols from their temporal logic specification. Since no efficient and programmable algorithm was known to us, we developed a new one to perform the translation of logic formulas into finite automata, based on the concept of derivatives (a la Brzozowski [Brz64]). We present the main results; proofs and examples can be found in [DJ88]. A similar technique is also used in [Gro89].

3.1 Derivatives of temporal formulas

Given a formula φ and a finite sequence s, the derivative of φ with respect to s is a formula $D_s\varphi$ such that :

$$\forall t \in \Sigma^*,\ t \models D_s\varphi \iff st \models \varphi$$

This defines a class of formulas which are all equivalent. $D_s\varphi$ represents any formula of that class.

Satisfaction may then be characterized by the emptyness acceptation of a derivative.

$$\forall s \in \Sigma^*, \forall \varphi \in TL,\ s \models \varphi \iff \lambda \models D_s\varphi$$

The derivative of a formula φ with respect to a finite sequence σ can be found recursively :

$$\begin{array}{lcl} \forall \alpha \in \Sigma, & D_{\sigma\alpha}\varphi & \equiv & D_\alpha D_\sigma \varphi \\ & D_\lambda \varphi & \equiv & \varphi \end{array}$$

If φ is a temporal formula, the derivative of φ with respect to a sequence α of unit length can be found recursively as follows :

$$\begin{array}{ll} D_\alpha \top \equiv \top & D_\alpha \alpha \equiv \top \\ D_\alpha \beta \equiv \bot\ (\forall \beta \in \Sigma,\ \alpha \neq \beta) & D_\alpha \neg\varphi \equiv \neg D_\alpha \varphi \\ D_\alpha(\varphi \wedge \psi) \equiv D_\alpha\varphi \wedge D_\alpha\psi & D_\alpha \bigcirc \varphi \equiv \neg(\neg\varphi \wedge \xi) \\ D_\alpha(\varphi \mathcal{U} \psi) \equiv \neg(\neg D_\alpha\psi \wedge \neg(D_\alpha\varphi \wedge \xi \wedge \varphi \mathcal{U} \psi)) & \end{array}$$

where

$$\xi = \neg(\bigwedge_{\alpha \in \Sigma} \neg\alpha)$$

3.2 Construction of the automaton accepting $L(\varphi, \Sigma)$

Two temporal formulas are said boolean-equivalent if they are equivalent according to the boolean calculus (one can derive the same representation using the boolean equivalence rules only). It does not imply equivalence, since two formulas may be equivalent but not recognized as boolean-equivalent.

Every formula φ has only a finite number of non-boolean-equivalent derivatives. All the distinct derivatives can be calculated considering sequences of increasing length.

From all the previous propositions, we can calculate the automaton $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_0, F)$:

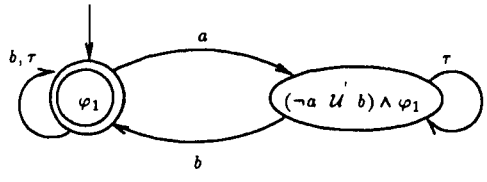
- The set of states Q is the finite set of the non-boolean-equivalent derivatives of φ,
- The transition function is determined by the existence of a derivative
  $\forall \psi, \psi' \in Q, \forall \alpha \in \Sigma,$
  $\psi' = \delta(\psi, \alpha) \Leftrightarrow \exists \sigma \in \Sigma^*,\ \psi \equiv D_\sigma\varphi \wedge \psi' \equiv D_\alpha\psi,$
- The initial state q_0 is φ, and a state ψ is a terminal state iff $\lambda \models \psi$.

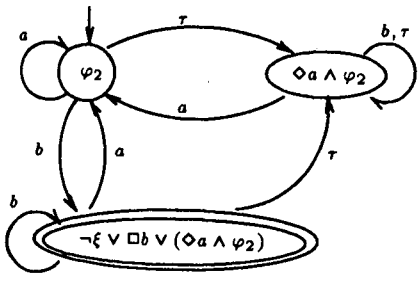
An algorithmic description is given by figure 3.

Figure 3: Text of the TL-compiler

```
type evt_type = scalar_type;
    state = record
            num : integer; — unique id of a state —
            fml : temporal_formula; — associated formula in a coded form —
            term : boolean; — true if terminal state —
        end;
    edge = record
            source, destination : integer;
            evt : evt_type; — label of the edge —
        end;
var Edges : set of edge init ∅;
    States : set of state init {(0, φ, λ ⊨ φ)};
    number : integer init 1;
function exists_boolean_equivalent (f: temporal_formula; var q : state ) : boolean ;
    — 'samé refers the same syntactic structure —
    — 'boolean_decision' is a boolean decision procedure —
    function boolean_equivalent (f,f' :temporal_formula): boolean ;
        begin
            if same(f,f') then boolean_equivalent := true
            else if λ ⊨ f and same(f',f ∨ ¬ξ) then boolean_equivalent := true
            else boolean_equivalent := boolean_decision(f ≡ f')
        end
    begin
        exists_boolean_equivalent := false;
        all x in States do
            if boolean_equivalent(f,x.fml) then begin
                exists_boolean_equivalent := true ;q := x end
    end
procedure succs (d : state );
    var y : temporal_formula; q : state; a : edge;
    begin
        all e in E do begin
            y := De(d.fml);
            if not (y ≡ ⊥) then begin
                if not exists_boolean_equivalent(y,q) then begin
                    q.num := number; q.fml := y; q.term := λ ⊨ y;
                    States := States ∪ { q } ;
                    number := number +1; succs(q) end;
                a.source := d.num; a.destination := q.num; a.evt := e;
                Edges := Edges ∪ { a } end
        end
    end
main
    succs((0, φ, λ ⊨ φ))
```

Figure 4: $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$





3.3 A TL-compiler

In order to illustrate the derivation process, we give in figure 4 the automata and derivatives of the examples φ_1 and φ_2.

The algorithm above has been implemented as a software package written in Pascal. Temporal logic formulas (where or- and and-formulas are considered as n-ary) are represented by trees.

Derivatives of some pure temporal formulas are kept during the computation to avoid re-derivation of previous terms, since the derivation rules can produce trees having common parts.

A lot of time is spent to test boolean-equivalence. In order to improve this procedure, we generate for each formula, the associated disjonctive normal form. Testing equivalence between normal forms is then straightforward.

The final program is 2500 lines long. With formulas having highly overlapped temporal operators, its performances fall down. But with usual properties, which are generally given as a conjonction of small formulas, it produces automata for about fifty temporal operators formulas in a few seconds on a SUN workstation.

4 Searching transitions and checking

4.1 State searching

According to our "on-line" approach, we try to search all the computations of the considered finite state system without recording the whole state space.

This is possible if we detect the loops. A general way to achieve loop detection with a limited memory is to use a depth-first strategy. We can then reach all the states using only a memory bounded with the state graph diameter.

The loop detection algorithm requires to keep the states of the current computation path. The others may be replaced if necessary.

Let us note that in general there does not exist a continuous function linking the memory size with the number of reached states [PJ88] (as illustrated in section 4.3).

Different replacement strategies can be applied in order to speed up the search. G. Holzmann studied this problem in [Hol87] : he found that random selection among the states to be deleted was the best management!

4.2 Checking

Let T_S and T_φ be the sets of fireable transitions (and the associated actions) in a given state (s_0, q_0) for the transition system $\mathcal{S}$ and the automaton associated to a temporal formula φ respectively

Figure 5: The model checker algorithm

```
stack := ∅; heap := ∅;
push(q_0^S, q_0^φ, enabled^S(q_0^S));
while stack ≠ ∅ do
 begin
  if top.enabled ≠ ∅ then
   begin t := one_element_of(top.enabled);
    top.enabled := top.enabled − [t];
    q^S := δ^S(top.state^S, t);
    q^φ := δ^φ(top.state^φ, t);
    if ¬((q^S, q^φ) ∈ stack) then
     if ¬((q^S, q^φ) ∈ heap) then
      begin T_φ := enabled^φ(q^φ);
       T_S := enabled^S(q^S);
       if ¬ [∀x ∈ T_S | (x ∈ T_φ) ∧ (δ^φ(q^φ, x) ∈ F)] then
        error
       else push(q^S, q^φ, T_S)
      end
   end
  else (* top.enabled = ∅ *)
   begin
    if full(heap) then
     replace(random(heap), top.state^S, top.state^φ)
    else memorize(top.state^S, top.state^φ);
    pop (* backtrack *)
   end;
 end;
end;
```

$(T_S \subseteq \Sigma, T_\varphi \subseteq \Sigma)$.

Since all the states of S are terminal states, the satisfaction relation can be reformulated as follows :

$$S \models \varphi \ \ iff \ \ T_S \subseteq T_\varphi \ \ and \ \forall \alpha \in T_S, \ \ \delta(q_0, \alpha) \in F$$

where q_0, δ, F refer to the automaton $\mathcal{A}_\varphi$. Considering the last relation, one can remark that we only need to generate the terminal states of the automaton descending from the initial state by only terminal states. This could be done directly in the TL-compiler. The latter relation would then be :

$$S \models \varphi \ \ iff \ \ T_S \subseteq T_\varphi$$

This condition can be evaluated during the search and then performs on-line checking. An algorithmic description is provided in figure 5. We explicitely manage a heap of already generated pairs (system state, automaton state) and a stack containing the triples (system state, automaton state, not yet fired transitions) of the current path. When we want to memorize a pair in the heap, if it is full, we make a random replacement.

4.3 Example

The example we have chosen is a very simplified connect-disconnect protocol (see figure 6) which was designed to provide an introduction to protocol validation [JR87].

We can see its state graph in figure 7 in the case where the channels are not bounded.

We are now concerned with bounded channels to have finite systems, so we will only refer to the part of the state graph over the line $n = constant$ where n is the size of the channel $A \rightarrow B$. We can easily verify that the number of different states is then $D = [\frac{n^2}{4}] + 3(n+1)$.

If we run the program with the formula *true*, which automaton is a single terminal state, then we generate all the states of the system. For a given n, we can modulate the heap size (HS) to observe the variations in number of generated states (NGS) and in time (T) according to that size. The curves are of course not continuous but we can draw their approximate behaviours (see figure 8). Those figures are only indications because the program has not been optimized (no particular coding of the states) and it is run on a very simple protocol.

However we can make the following remarks:

- If the heap size HS is close to D then also is NGS because we travel through a minimum of already generated states. Though the larger is the heap, the longer we spend time searching if it contains the last generated state. So, the time of successive states generation is very short in comparison with the time spent searching.

- If HS comes near zero, then we almost only detect loops and we regenerate a lot of states. So NGS explode and the time spent in generation starts prevailing over that of searching. This one however stays very important because, although each search is cheap, it is performed very often.

- The best results are obtained for intermediate sizes. When starting from D we decrease HS, then NGS stays for a long time very close to D, just because of the random replacement. So generation is cheap, NGS is small and the reasonable size of the heap allows relatively rapid searches.

  In our example, those best results are obtained with a heap size near $\frac{D}{10}$. The time diminution is then half in comparison with the case where $HS \simeq D$.

Those observations can of course also be done with other formulas for which the number of generated states only increase when loops of the automaton include some of the state graph.

To conclude we would say that, if we dispose of a sufficient memory to keep all the different states, then we can improve with a smallest one. So, with a given memory size, we can very

Figure 6: The connect-disconnect protocol

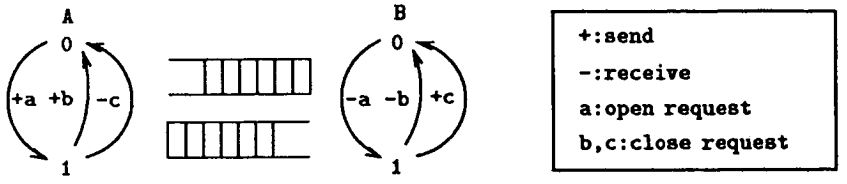


Figure 7: State graph with unbounded channels

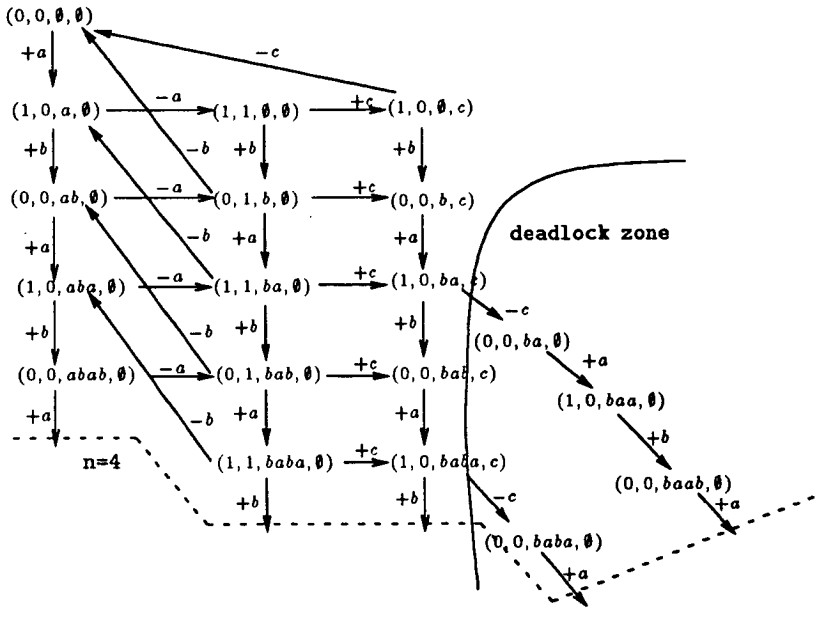
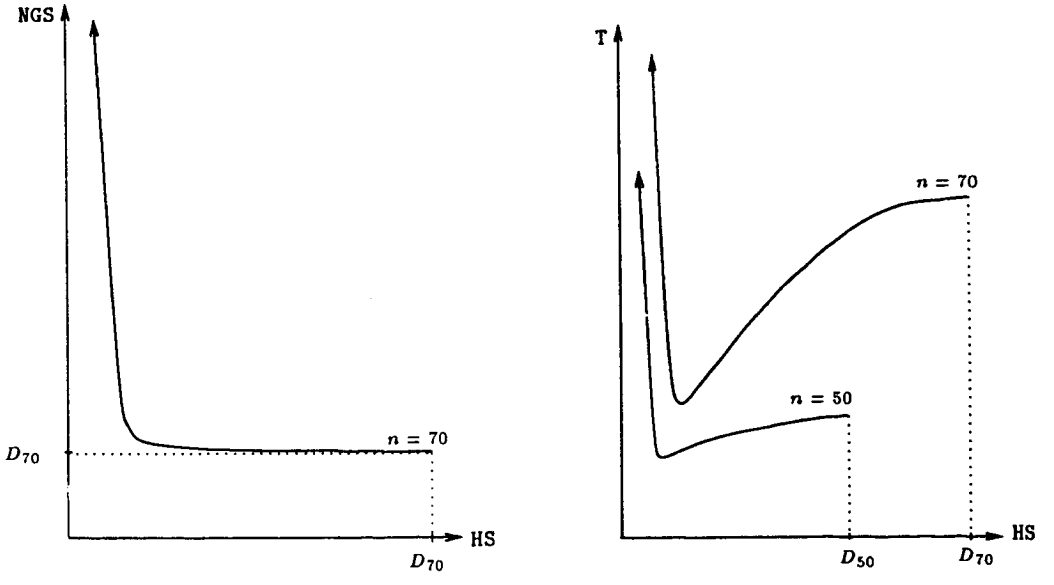


Figure 8: Number of generated states and time in function of heap size



efficiently check protocols which couldn't be analysed with methods requiring generation of all the different states and then checking properties. When the checked formula is not valid on the state graph, we don't need to generate all the states, because we stop as soon as the formula becomes false.

We think that those observations can be generalized for a lot of protocols because, even if the curves are not the same, they certainly have a similar form.

5 Extension to the infinite computations

Our approach can easily be extended to some kinds of infinite computations. We give here some hints.

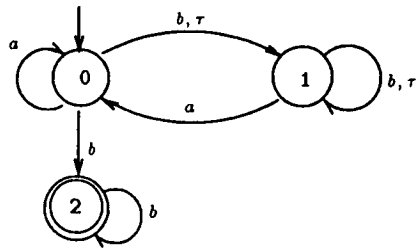
We consider properties given by deterministic Büchi automata (as advocated in [MP87]).

For such an automaton, an infinite sequence σ is accepted if and only if it spells out infinitely often a terminal state. Since we only consider finite state systems, it follows that the system satisfies the property if and only if every fireable transition of the system is also a transition of the Büchi automaton and if every elementary cycle of the state graph is valued by a terminal state of the automaton.

The checking algorithm is basically the same as for the finite case except that an error occurs when a loop is detected and all its states are valued by non-terminal states of the automaton.

Application to temporal logic formulas is more disputable. We know that a temporal formula does not ever correspond to a deterministic Büchi automaton. For exemple, the formula φ_2 is not deterministic in that sense (figure 9 gives the associated and irreducible non-determinisic Büchi automaton).

Figure 9: The non-deterministic automaton of φ_2



If a formula φ is deterministic, our translation algorithm gives the right deterministic Büchi automaton $\mathcal{A}_\varphi$. To efficiently decide the determinism of a formula is yet an open problem.

6 Conclusion

Avoiding state space explosion in model-checking algorithms is a good challenge to improve the applicability of verification tools.

We have presented, in that context, an approach called on-line model-checking where satisfiability is checked during the state generation process.

Though the entire validation has to be rerun for each new property, this approach is interesting since it decreases the state space needed (which may theoretically be decreased to the state graph diameter).

We also have shown that for large graphs, surprisingly the on-line technique may be better in time than the classical model-checking. Precise algorithms are given and they have been systematically experimented.

Nevertheless, we have only dealt with a simple context, considering finite computations and a basic linear temporal logic. Future works could be to transport the idea towards branching time logics (may be difficult) and to find efficient algorithms to check satisfiability of linear formulas on infinite computations (accessible).