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

Claude JARD, Thierry JERON
ADP research team, IRISA, Campus de Beaulieu
F-35042 RENNES Cedex FRANCE
jard@irisa.fr, jeron@irisa.fr

1 Introduction

If we restrict our attention to finite state programs (variables and communication channels if any range over finite domains), then the whole program can be represented as a (generally large) finite graph. 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 (LTL) and atomic propositions as actions [16]. Our terminology is no essential restriction and simplifies the transition to the automata framework we use thereafter.

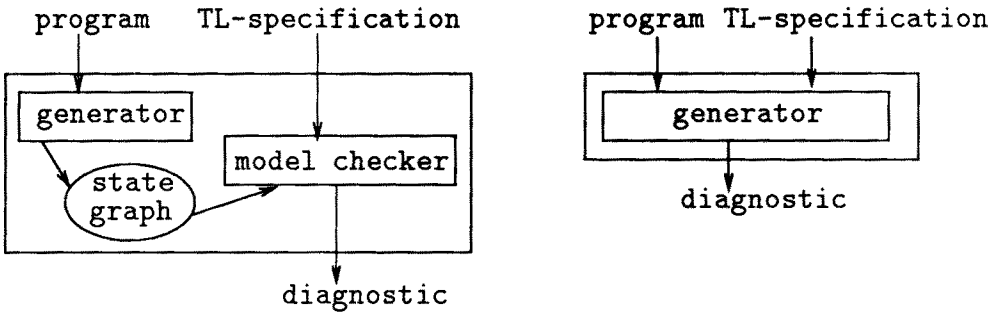
Models for linear logic are totally ordered computations. We restrict our attention to finite computations. Extension to the infinite case will be discussed.

Classical model-checking as implemented in EMC [2] or XESAR [14] (for a branching time temporal logic) is illustrated in the left part of 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. 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 [3] for a branching time logic.

Our paper presents a first step to a complementary approach that we call "on-line model-checking" and which tends to considerably decrease the state space needed. *The basic idea is to check during the state enumeration* (see figure 1). For that aim, the temporal logic specification must be executed [15]. The logic specification is translated into a finite automaton (here is the main algorithmical difficulty) which will value the system states during enumeration. Decision of validity or rejection can then be reached in finite time providing a large enough memory to store a number of states equal to the depth of the state graph. A related technique is formally described in [8].

In section 2 we present the considered temporal logic. In the third part, we recall an effective algorithm we designed to translate logic formulas into finite automata when they are interpreted over finite computations. Section 4 exposes the model-checking algorithm

Figure 1: A classical verifier and an on-line verifier



and its properties. We then discuss an extension to the infinite case for a particular (but rather large) class of formulas, called "deterministic".

2 The considered TL

Let Σ be a finite alphabet (the set of observable events, augmented with the invisible action τ). The temporal formulas over Σ ($\mathcal{F}$) are built up from the atomic propositions $\alpha \in \Sigma$ using the boolean connectives $\neg$, $\wedge$, the unary temporal operator $\bigcirc$ ("next"), the binary temporal operator $\mathcal{U}$ ("until"), and brackets [13]. 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 ($\Sigma = \{a, b, \tau\}$): $\varphi_1 = \Box(a \Rightarrow \bigodot(\neg a \mathcal{U} b))$ *and* $\varphi_2 = \Diamond a \ \mathcal{U} \ \Box b$

The formulas are interpreted over finite nonempty sequences $\sigma = \sigma_0 ... \sigma_n$ ($n \geq 1$) which length is $\mid \sigma \mid$.

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

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

$(\sigma, i) \models \top$ *always*, $\quad (\sigma, i) \models \neg\varphi$ *iff* *not* $((\sigma, i) \models \varphi)$

$(\sigma, i) \models \alpha$ *iff* $\sigma_i = \alpha$, $\quad (\sigma, i) \models \varphi \wedge \psi$ *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* $(\forall k,\ i \leq k < j,\ (\sigma, k) \models \varphi)$

We say that a computation σ satisfies a formula φ ($\sigma \models \varphi$) iff $(\sigma, 0) \models \varphi$. 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 Σ defines a set $L(\varphi, \Sigma)$ of finite sequences s.t. :

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

Examples: $L(\varphi_1, \Sigma) = (b \cup \tau)^* \cup (a\tau^* b(b \cup \tau)^*)^*$ *and* $L(\varphi_2, \Sigma) = b^+ \cup \Sigma^* a b^+$

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. The associated finitary language over an alphabet Σ is the set of all the possible computations (ϕ renames the invisible actions $E - \Sigma$ into τ)

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

A transition system satisfies φ iff all its finite computations on Σ satisfy φ :

$$\mathcal{S} \models \varphi \;\; iff \;\; \forall \sigma \in L(\mathcal{S}, \Sigma), \;\; \sigma \models \varphi \;\; (\; i.e. \; L(\mathcal{S}, \Sigma) \subseteq L(\varphi, \Sigma)$$

3 From TL to finite automata

The theory of linear time logic was linked to the automata theory several years ago. We know that a language is TL-definable if and only if it is first-order definable, and that first-order definability can be characterized elegantly in terms of "star-free" languages (regular languages included in the closure of the finite word-sets under concatenation and boolean operations only) [9].

Examples (Σ^* is an abbreviation for $\neg\emptyset$) :

$$L(\varphi_1, \Sigma) = \neg[\Sigma^* a \neg(\Sigma^* b \Sigma^*) \cup \Sigma^* a \neg(\Sigma^* b \Sigma^*) a \Sigma^*]$$
$$L(\varphi_2, \Sigma) = b \neg(\Sigma^*(a \cup \tau)\Sigma^*) \cup \Sigma^* ab \neg(\Sigma^*(a \cup \tau)\Sigma^*)$$

The above proposition was first applied in [11] to show the possibility of synthesizing synchronization skeletons of protocols from their TL-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 [1]). We present the main results; proofs and examples can be found in [4]. A similar technique is also used in [5].

3.1 Derivatives of temporal formulas

A derivative of a formula φ with respect to the finite sequence s is a formula $D_s\varphi$ s.t. :

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

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

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

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

$$\forall \alpha \in \Sigma, D_{\sigma\alpha}\varphi \equiv D_\alpha D_\sigma \varphi \;\; and \;\; D_\lambda \varphi \equiv \varphi$$

The derivative of φ with respect to a sequence α of unit length can be found recursively :

$$D_\alpha \top \equiv \top \qquad\qquad D_\alpha \alpha \equiv \top$$
$$D_\alpha \beta \equiv \bot \; (\forall \beta \in \Sigma, \; \alpha \neq \beta) \qquad D_\alpha \neg\varphi \equiv \neg D_\alpha \varphi$$
$$D_\alpha(\varphi \wedge \psi) \equiv D_\alpha\varphi \wedge D_\alpha\psi \qquad D_\alpha \bigcirc \varphi \equiv \neg(\neg\varphi \wedge \xi)$$
$$D_\alpha(\varphi \mathcal{U} \psi) \equiv \neg(\neg D_\alpha\psi \wedge \neg(D_\alpha\varphi \wedge \xi \wedge \varphi \mathcal{U} \psi)) \qquad where \; \xi = \neg(\textstyle\bigwedge_{\alpha \in \Sigma} \neg\alpha)$$

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

Two temporal formulas are said boolean-equivalent (does not imply equivalence) if they are equivalent according to the boolean calculus only.

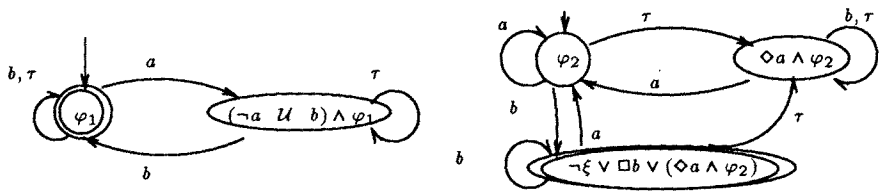
Every formula φ has only a finite number of non-boolean-equivalent derivatives. All the distinct derivatives can be calculated considering sequences of increasing length.

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

- 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 $\lambda \models \psi$.

In order to illustrate the derivation process, we give in figure 2 the automata and derivatives of the examples φ_1 and φ_2.

Figure 2: $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$



The TL-compiler has been implemented as a software package written in Pascal (2500 lines) and ,with usual properties, produces automata for about fifty temporal operators formulas in a few seconds on a SUN workstation. TL formulas are represented by trees (or- and and-formulas are considered as n–ary). Derivatives of some pure temporal sub–formulas are kept during the computation to avoid re–derivation of previous terms, since the derivation rules can produce common sub–trees. A conjonctive normal form is generated in order to improve the boolean–equivalence testing.

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 by using a depth-first strategy and keeping the states of the current computation path (the others may be replaced if necessary). We can then theoretically reach all the states using only a memory bounded with the state graph diameter. Let us note that in general there does not exist a continuous function linking the memory size with the number of reached states [12] (as illustrated in section 4.3). Different replacement strategies can be applied in order to speed up the search. Holzmann studied replacement strategies in order to speed up the search [6] : he found that random selection among the states to be deleted was the best management!

Figure 3: The model checker algorithm

```
stack := ∅; heap := ∅;
push(q0^S, q0^φ, enabled^S(q0^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 begin (* top.enabled = ∅ *)
    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;
```

4.2 Checking

Let $T_S \subseteq \Sigma$ and $T_\varphi \subseteq \Sigma$ be the sets of fireable transitions in a given state (s_0, q_0) for the transition system $\mathcal{S}$ and the automaton associated to a temporal formula φ respectively. Since all the states of $\mathcal{S}$ are terminal states, the satisfaction relation can be reformulated as follows :

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

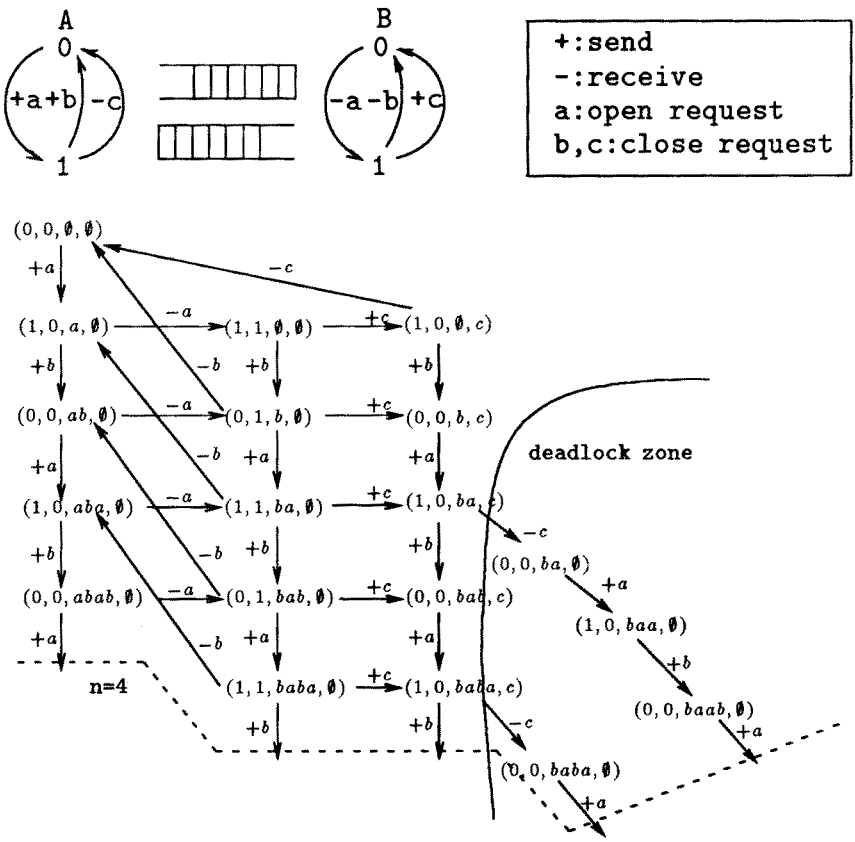
where q_0, δ, F refer to the automaton $\mathcal{A}_\varphi$. This condition can be evaluated during the search and then performs on-line checking. In the algorithm (see figure 3) we explicitely manage a heap (with random replacement) of already generated pairs (system state, automaton state) and a stack of triples (system state, automaton state, not yet fired transitions) of the current path.

4.3 Example

The example is a very simplified connect-disconnect protocol (see figure 4) which was designed to provide an introduction to protocol validation [7]. Being concerned with finite systems, we will only refer to the part of the state graph over the line $n = constant$ where n is the size of the channel $A \longrightarrow B$ (the number of different states is then $N = [\frac{n^2}{4}] + 3(n+1)$).

If we run the program with the formula *true*, which automaton is a single terminal state, 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

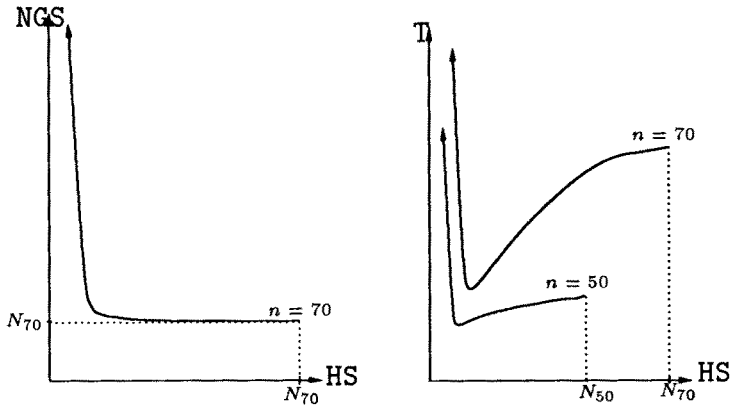
Figure 4: The connect-disconnect protocol and its infinite state graph



T according to that size (see figure 5). 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 remark: when starting from HS $\simeq$ N, we decrease HS, NGS stays for a long time very close to N (before explosion) just because of the random replacement. So generation time is almost minimum and searching time decrease. In our example, the best results are obtained with a heap size near $\frac{N}{10}$. 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.

Hence, with a given memory size, we can very 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.

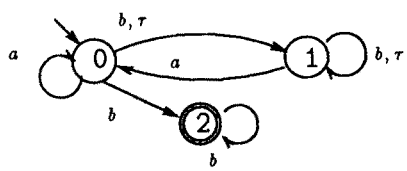
Figure 5: Number of generated states and time in function of heap size



5 Extension to the infinite computations

We give here some hints of the extension of our approach to some kinds of infinite computations. Considering a property given by a deterministic Büchi automata (as advocated in [10]), 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.

Figure 6: The irreducible non-deterministic Büchi automaton of φ_2



Application to temporal logic formulas is more disputable since a temporal formula does not ever correspond to a deterministic Büchi automaton (see figure 6). 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. 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).

References

[1] J. Brzozowski. Derivatives of regular expressions. *Journal of the Association of Computing Machinery*, 11(4):481–494, October 1964.

[2] E. Clarke., E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification : a practical approach. *ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing*, 117–126, 1983.

[3] E. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. 6^{th} *ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing, Vancouver, Canada*, August 1987.

[4] O. Drissi-Kaitouni and C. Jard. *Compiling temporal logic specifications into observers.* Technical Report 881, IRISA University of Rennes, July 1988.

[5] R. Groz. Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations. *Thèse de doctorat de l'universite de Rennes I, No 194*, January 1989.

[6] G. Holzmann. Automated protocol validation in argos, assertion proving and scatter searching. *IEEE trans. on Software Engineering, Vol 13, No 6*, June 1987.

[7] C. Jard and M. Raynal. *Specification of Properties is Required to Verify Distributed Algorithms.* Technical Report 651, INRIA, Centre IRISA, Rennes, February 1987.

[8] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. 12^{th} *Symposium on Principles of Programming Languages, Austin, Texas*, 97–107, 1984.

[9] R. Mac Naughton and S. Papert. Counter free automata. MIT Press, Cambridge, Avril 1971.

[10] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. 14^{th} *Symposium on Principles of Programming Languages, ACM, Munich*, 1–12, January 1987.

[11] Z. Manna and P. Wolper. Synthesis of communication processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–98, Junary 1984.

[12] J. Pageot and C. Jard. Experience in guiding simulation. *Protocol Specification, Testing, and Verification, VIII, IFIP*, 207–218, June 1988.

[13] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. *LNCS #224, Current Trends in Concurrency*, 510–584, 1986.

[14] J. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In 7^{th} *IFIP International Workshop on Protocol Specification, Testing, and Verification, Zurich, Suisse*, North Holland, May 1987.

[15] J. Sifakis. A response to amir pnueli's : specification and development of reactive systems. *IFIP'86, Dublin*, 1986.

[16] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Control*, 48(3):261–283, 1981.