Bounded-memory Algorithms for Verification On-the-fly *

Claude JARD, Thierry JÉRON

IRISA, Campus de Beaulieu
F-35042 RENNES Cedex FRANCE
jard@irisa.fr, jeron@irisa.fr

Abstract

One of the main limitations of today's verification tools is the size of the memory needed to exhaustively build the state graphs of the programs. But for numerous properties, it is not necessary to explicitly build this graph and an exhaustive depth–first traversal is often sufficient. In order to avoid retraversing states, we must then store some already visited states in memory. But since the memory size is bounded, visited states must be randomly replaced. In most cases this depth–first traversal with replacement can push back the limits of verification tools.

1 Introduction

1.1 Motivation

Program verification is a branch of computer science whose business is "to prove programs correctness". It has been studied in theoretical computer science departments for a long time but it is rarely and laboriously applied to real world problems. As a matter of fact, we must pay much more attention to practical problems like the amount of space and time needed to perform verification. Let us recall that proofs of correctness are proofs of the relative consistency between two formal specifications: those of the program, and of the properties that the program is supposed to satisfy. Such a formal proof tries to increase the confidence that a computer system will make it right when executing the program under consideration.

A considerable need for such methods appeared these last ten years in different domains, such as design of asynchronous circuits, communication protocols

*Claude Jard is a full time researcher at the Centre National de la Recherche Scientifique (CNRS), Thierry Jéron is a PhD student at the University of Rennes. This work was also funded by the french national project C^3 on parallelism.

and distributed software in general. A lot of us accepted the challenge to design automated verification tools, and many different theories have been suggested for the automated analysis of distributed systems. There now exist elaborate methods that can verify quite subtle behaviors.

A simple method for performing automated verification is symbolic execution which is the core of most existing and planned verification systems. The practical limits of this method are the size of the state space and the time it may take to inspect all reachable states in this state space. Those quantities can dramatically rise with the problem size.

1.2 Current state-of-art

Reachability analysis is basically an exhaustive search yielding a rooted graph of global states. This technique is often called *perturbation* [Wes78]. Starting from some specified initial state, successor states are generated and stored in the computer. The process stops when no new state (i.e. one not previously stored) can be generated. Termination is guaranteed if all the program variables (including communication channels) are bounded.

The state graph is usually very large and for example, any protocol of practical relevance will have a state space in the order of one million states. There are two major problems when handling systems of this size: state matching (to avoid double work and to ensure termination), and state storing.

We will suppose that the memory is arranged as a balanced tree, that reachable states are numbered from 1 to R, and that states are of constant size S. The memory size M needed is then at least $R.S$. Let $C(S)$ be the time needed for the comparison of two states. The first time a state i is generated, the memory contains $i-1$ states, thus its insertion in the tree is carried out in time at worst $C(S).\log(i)$. If d is the average degree of nodes, each node is re–generated $d-1$ times and searched in a memory which contains at least i states. The time needed for those searches can be approximated by $(d-1).C(S).\log(i)$. Coarsely approximating $\log(R!)$ by $R.\log(R)$, we say that the time complexity of the perturbation technique is

$$T \simeq d.C(S).\Sigma_{i=1}^{R} \log(i) \simeq d.C(S).\log(R!) \simeq d.C(S).R.\log(R)$$

If $M = 10^7$ bytes and $S = 10^2$ bytes, the size of the graphs that can actually be analysed is less than $R = 10^5$ states. If $d = 2$, $C(S) = 10^{-4}$ seconds, and trees are binary trees, the time needed is $T \simeq 6$ minutes.

In order to master the "state explosion", different works have been conducted to reduce the size of the graph [CG87, Val90, BFH90, GS90, GW91]. Obviously, reduction must be performed during the graph generation. The other constraint is that the validity of properties to be verified must not be changed. For that reason, we do not consider simulation methods which provide only partial verification [JGM85, Hol85, Wes86, PJ88, JGM88, Hol89].

1.3 Verification on-the-fly

The key idea is that, for a large class of properties, storing all the reachability graph is not mandatory. It is enough to visit all the states and/or all the transitions. A depth-first traversal of the reachability graph performs such an exhaustive search. Only the current path has to be stored but the time needed to perform a verification can be catastrophic, due to the re-generation of forgotten states.

We propose an intermediate method which offers a good compromise between time and space requirements. It is based on a depth-first traversal but uses all the available space in order to store not only the current path, but also the greatest possible number of already visited states. We will prove that bounding memory to a smaller size than the state space may not significantly increase the time complexity. Such algorithms allow us to build efficient verifiers, able to handle large graphs. This approach is often called "verification on-the-fly".

It was first proposed in terms of "on-line model-checking" by the authors in [JJ89]. Since then, similar ideas have been advocated in [CVWY90] and [FM90]. [CVWY90] presents efficient algorithms to compare Büchi automata and thus proposes a new solution to the verification of temporal properties on infinite behaviors of finite state programs. [FM90] extends the technique to verify on-the-fly bisimulation equivalences on transitions graphs. The core of the method is to traverse (during its generation) a kind of product of finite transitions systems. Unifying these different views would be an interesting prospect.

The remainder of the paper is organized as follows. We present in detail a class of bounded-memory algorithms that traverse exhaustively the state space of the program to be verified. Upper bounds for space and time complexities are computed and different experiments show the average behavior of our algorithms. Another part of the paper discusses applications, namely verification of safety properties and testing unboundedness of Fifo channels. We conclude with some prospects.

2 Depth-first traversal with replacement

We saw above that the main drawback of a perturbation technique is the memory size needed to perform the graph generation of real size systems. Now, there are some verifications for which a traversal of all states and transitions is enough. It is then unnecessary to store the whole graph. An algorithm performing this exhaustive traversal is a depth-first traversal in which we theoretically only need to detect cycles, provided that the memory is large enough to store the longest acyclic sequence. Unfortunately, visited states which no longer behave to the current sequence are forgotten and can be visited again in many other sequences. In the best case the number R_{gen} of generated states is R. But in the worst case R_{gen} can reach $R!.e$ for a complete graph with R states (e is the basis of natural logarithms). If the number of states in the memory is bounded by the length of the longest acyclic sequence D_{max}, the time needed to complete the traversal is in the scale

$$C(S).R.\log(D_{max}) \leq T \leq C(S).R!.e.\log(D_{max})$$

However, a depth–first traversal can significantly be improved if the whole memory amount is used [J9́1a]. Actually, since $D_{max}.S < M$, one can use the remainder of the memory to store already visited states, and then avoiding re-traversing some states. We present this technique and show with examples that it can be efficiently used to analyse real size graphs which are too large to fit in memory.

2.1 The algorithm

```
St_Stack:=nil; (* -- states of the current sequence -- *)
Tr_Stack:=nil; (* -- stack of sets of pending transitions -- *)
Visited:=∅; (* -- already visited states -- *)
push (S0, St_Stack); push(fireable(S0),Tr_Stack);
while St_Stack ≠ ∅ do begin
  S:=top(St_Stack); (* -- current state -- *)
  if top(Tr_Stack) ≠ ∅ then begin
    t:=extract_one_elt_of(top(Tr_Stack)); (* -- choose and remove -- *)
    S':=succ(S,t);
    if S' ∉ St_Stack ∪ Visited then begin
      if memory_full then begin (* -- replacement -- *)
        S_del:=one_state_from(Visited);
        Visited:=Visited - {S_del};
        end;
      push(S',St_Stack);
      push(fireable(S'),Tr_Stack);
      end;
    end
  else begin (* -- top(Tr_Stack) = ∅ -- *)
    pop(St_Stack);
    pop(Tr_Stack);
    Visited:=Visited ∪{S};
    end;
  end;
```

An algorithm performing a depth–first traversal with replacement is described above. It is very similar to a classical depth–first traversal except for the set $Visited$ of already visited states and the execution phase when the memory is full.

The traversal with replacement algorithm can be used on every graph such that $D_{max}.S \leq M$. But, contrarily to the simple traversal, it is not a necessary condition for the termination because states of the longest acyclic sequences can be reached by shortest sequences. A necessary condition is $G_{max}.S \leq M$ where G_{max} is the maximal length of a geodesic with initial state S_0 (a geodesic from S to S' is an acyclic sequence from u to v with minimum length). We have $G_{max} \leq D_{max}$ but if $G_{max}.S \leq M \leq D_{max}.S$ the algorithm may or may not terminate, depending on the order of transitions evaluations.

2.2 Time complexity

Let us remark that we always have $(|St\_Stack| + |Visited|).S \leq M$ and the boolean variable $Memory\_full$ is equal to $(|St\_Stack| + |Visited|).S = M$ and is a stable property. Let R_{ins} be the number of insertions of states in the memory i.e. $St\_Stack \cup Visited$. The behavior of the algorithm in the case $R.S \leq M$ is almost the same as a perturbation, except for the generation order. Each state is inserted once, so $R_{ins} = R$. The time complexity is then approximately the same.

Now if $R.S > M$, R_{ins} exceeds R because an already visited state may have been forgotten. Due to the stability of the property $memory\_full$, we can separate the algorithm into two phases:

- in the first phase, when $\neg memory\_full$, all visited states are in $St\_Stack \cup Visited$ and the algorithm behaves like a perturbation,
- in the second phase, when $memory\_full$, each time a state S' is generated and not found in $St\_Stack \cup Visited$, we must remove one state S_{del} from $Visited$ before pushing S' in St_Stack. The way this replacement is performed influences the total number of generated states R_{gen}.

We also suppose that the whole memory $St\_Stack \cup Visited$ is arranged as a balanced tree, which supports access, insertion and deletion operations in logarithmic worst case. The number of states in that memory is always less than M/S. Each generated state must be searched in that memory. Thus, the total time of the traversal is approximately

$$T \simeq C(S).R_{gen}.\log(\frac{M}{S})$$

Recall that for the perturbation, time complexity is $C(S).d.R.\log(R)$. If $M \leq R.S$, we have $R_{gen} \simeq d.R$, thus complexities are identical. If $M > R.S$, a perturbation technique is no longer possible. We have $\log(M/S) < \log(R)$ thus, if R_{gen} is in the same order of magnitude that $d.R$, time complexity of the depth–first traversal is close to the complexity that a perturbation would have with a memory of size $R.S$.

The relation between R_{gen} and R_{ins} is almost the same that the one between $d.R$ and R. So, we expect that R_{ins} will stay close to R.

The choice of a replacement strategy is then essential in such an algorithm. Several strategies have been looked at. As noticed in [Hol87], the best one seems to be random replacement. It is easily performed and has no performance drop for particular graphs.

2.3 Experiments

The depth–first traversal with replacement has been used with different kinds of graphs. Some of them are accessibility graphs of communication protocols modelled by communicating finite state machines, and others are random graphs. The parameters of these random graphs are R_{max} a bound on the number of states and d_{max} the maximum degree of a node. They are generated in a breadth–first way. The degree of each node is chosen uniformly between 0 and d_{max}. If g is the number of already generated states, each successor of the current state has probability $1-g/min(2.g, R_{max})$ to be a new state. Among those random graphs, we only considered those with R close to R_{max}.

The two curves of figure 1 represent the behavior of the algorithm on a random graph when decreasing the memory size. Starting from $M_{max} = R.S$, the memory size is decreased down to the minimal possible value M_{min} for which the algorithm terminates. The two bounds M_{max}/S and M_{min}/S are figured by the two dashed vertical lines.

The two first curves represent the evolution of the number of insertions R_{ins} of states in $St\_Stack \cup Visited$ and the execution time. If $M = M_{max} = R.S$ then $R_{ins} = R$. When M decreases, R_{ins} increases. But it increases very slowly until M comes very near from M_{min}. The number R_{ins} is then less than twice R. Finally R_{ins} explodes but the memory has been significantly reduced before explosion. The execution time T has then a similar form. For that example, with a memory size of 40% of $R.S$ we have only 70% more insertion of states, which give only 50% time increase.

Many examples have been tested with this traversal. They almost gave the same type of curves. But we can only decrease the memory size downto a value between $D_{max}.S$ and $G_{max}.S$. Thus when D_{max} is small with respect to R, one can reduce M very significantly, and the increase of R_{ins} and T are very slow. In the left example of figure 2, $M_{min} = M_{max}/10$, and we have an increase of only 1% of R_{ins} and 11% of T (see the left hand curve of figure 2). However, for graphs in which D_{max} is close to R as in the right hand curve of figure 2, that is when graphs are very connected (a complete graph is the worst case) results are not so good. The domain in which M/S can vary is very tight and R_{ins} and T increase very quickly.

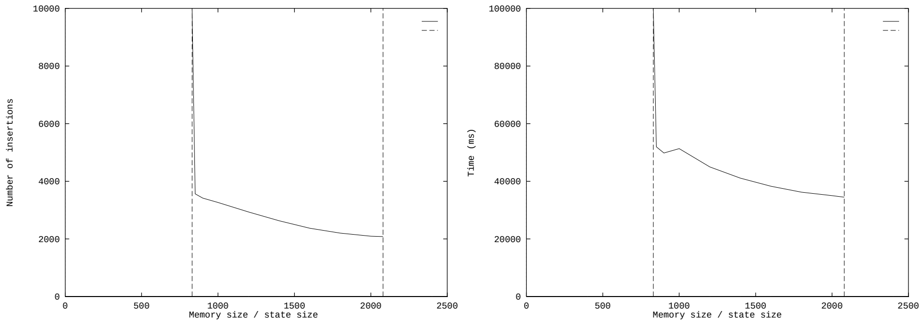

Figure 1: Number of generated states and execution time

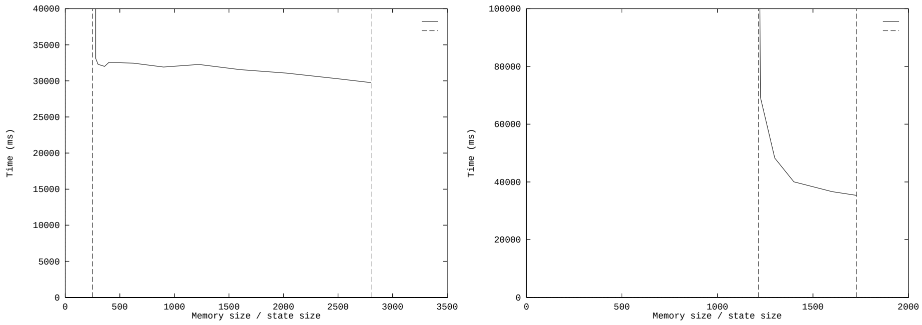

Figure 2: Execution time in extremal cases

3 Some applications

3.1 On–line model–checking

The first application of the depth–first traversal with replacement was introduced in [JJ89]. The purpose was to verify that a protocol specification satisfied a property f. Properties were expressed in event based linear temporal logic (LTL) [Pnu86] and translated into finite states automata. But the idea can be generalized to other formalisms such as Büchi automata.

Let $\mathcal{S}_1 =< Q_1, A, T_1, q_{0_1} >$ be the labelled transition system associated to the specification *Spec* where Q_1 is a finite set of states, A a finite set of actions, $T \subseteq Q_1 \times A \times Q_2$ the transition relation, and q_{0_1} the initial state.

Suppose that a property $\mathcal{P}$ can be expressed by a deterministic Büchi automaton $\mathcal{B} =< Q_2, A, T_2, q_{0_2}, F_2 >$ where Q_2 is its finite set of states, A its set of actions, $T_2 \subseteq Q_2 \times A \times Q_2$ its transition relation, q_{0_2} the initial state and F_2 a set of designated states. An infinite word $a_1 \ldots a_n \ldots \in A^\omega$ is recognized by $\mathcal{B}$ if and only if there exists an infinite run of $\mathcal{B}$: $q_{0_2} \stackrel{a_1}{\to} q_1 \ldots q_{n-1} \stackrel{a_n}{\to} q_n \ldots$ such that $q_i \in F_2$ for infinitely many i's.

We say that *Spec* satisfies $\mathcal{P}$ written $Spec \models \mathcal{P}$ if and only if every infinite word labelling an infinite transition sequence of $\mathcal{S}_1$ is recognized by $\mathcal{B}$.

When the Büchi automaton is not necessarily deterministic, the usual way to verify that $Spec \models \mathcal{P}$ is to consider $\mathcal{S}_1$ as a Büchi automaton (its set of designated states is Q_1), make the product of $\mathcal{S}_1$ with the complement automaton $\overline{\mathcal{B}}$ of $\mathcal{B}$ and check if $\mathcal{S}_1 \times \overline{\mathcal{B}}$ is empty (accepts no word). This can be done by computing the strongly connected components.

In the case of a deterministic Büchi automaton, we will show that there is a very simple algorithm which performs this verification without complementation and without computation of strongly connected components [J9́1a].

We consider $\mathcal{S}_1$ as a Büchi automaton with Q_1 as its set of designated states. We suppose that $\mathcal{B}$ is complete. This can always be done by adding a new state.

The synchronous product of $\mathcal{S}_1$ and $\mathcal{B}$ is $\mathcal{S} =< Q, A, T, q_0, F >$ with:

- $Q = Q_1 \times Q_2$,
- $q_0 = (q_{0_1}, q_{0_2})$,
- $F = Q_1 \times F_2$,
- $T \subseteq Q \times A \times Q$ is defined by

  $((q_1, q_2), a, (q_1', q_2')) \in T$ if and only if $(q_1, a, q_1') \in T_1$ and $(q_2, a, q_2') \in T_2$

Since $\mathcal{B}$ is complete, the infinite sequences of executable actions of $\mathcal{S}_1$ are exactly the words labelling the infinite runs of $\mathcal{S}$. And according to the definition of $\mathcal{S}$, $Spec \models \mathcal{P}$ if and only if every infinite run of $\mathcal{S}$ contains infinitely many states of F. Considering $\mathcal{S}$ as a directed graph, it is equivalent to say that every cycle of the graph contains a vertice in F. But this is equivalent to say that the sub-graph $\mathcal{S}'$ obtained from $\mathcal{S}$ by removing all vertices of F (and the correponding edges) is acyclic. And $\mathcal{S}'$ is acyclic if and only if a depth-first traversal of $\mathcal{S}'$ doesn't detect any cycle.

But we don't want to first build $\mathcal{S}$ and then remove vertices of F. We would like to check whether that $\mathcal{S}'$ is acyclic during a traversal of $\mathcal{S}$. This can be done by a traversal of $\mathcal{S}$ which is composed of depth-first traversals of sub-graphs of $\mathcal{S}$ in the following way. We need a set W initialized with $\{q_0\}$ which contains the roots of the depth-first traversals not yet performed. These roots are q_0 and all states of F. For each traversal initiated in a state $q_{init} \in W$, remove q_{init} from W and add the following to the traversal with replacement: if a new state $q \in F$ is reached, it is added to W and successors of q are not explored now (they will be in the traversal initiated in q) and if a cycle is detected in $q \notin F$ this simply signifies that a cycle of $\mathcal{S}'$ is detected.

The algorithm stops when W is empty and $Spec \models \mathcal{P}$ if and only if no cycle of $\mathcal{S}'$ is detected.

The depth–first traversal with replacement is very efficient for the on–line model–checking of deterministic finite states and Büchi automata. It avoids constructing the complete reachability graph. Moreover, you can choose to stop as soon as an error is detected.

3.2 Bisimulation

In the paper [FM90], a related technique is used in order to verify strong bisimulation equivalence on the fly. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two transitions systems and $\mathcal{S}$ their synchronous product. If at least one of the transitions systems is deterministic, $\mathcal{S}_1$ and $\mathcal{S}_2$ are strongly bisimilar if and only if there is no transitions sequence of $\mathcal{S}$ leading to the state *fail*. It is very similar to the algorithm in [JJ89], and a depth–first traversal with replacement can be used and is certainly efficient.

If none of the two transitions systems is deterministic, verifying bisimulation equivalence on the fly is much more tricky. A pure depth–first traversal with a postfix analysis of states is possible but is certainly inefficient. Now, if visited states are stored, several runs of the depth–first traversal may be necessary to complete the analysis. However practical experiments showed that the number of depth-first traversals to be performed is weak.

3.3 Verification of temporal properties

In the paper [CVWY90], the authors define a method for the verification of a temporal logic property f on a finite state program P which combines a depth–first traversal with a partial search with hashing [Hol89]. It is based on the construction of the Büchi automaton which is equivalent to a linear temporal logic formula [VW86].

Let $A_{\neg f}$ be the Büchi automaton corresponding to the formula $\neg f$ and F its set of designated states. P is considered as a Büchi automaton with all states designated. P satisfies f if the product automaton $P \times A_{\neg f}$ accepts no sequence.

Instead of computing the strongly connected components of the automaton, the algorithm performs two depth–first traversals: the first one orders the reachable designated states of the product automaton and, using this ordered set, the second one determines if one of these designated states is in a cycle.

In order to reduce the memory requirements, visited states are hashed in a bit array of size m. The hash function h can then hash different states to the same bit, and consider them as equal [Hol89]. Thus, due to collisions, the algorithm can erroneously conclude that P satisfies f, but never erroneously concludes that P does not satisfy f.

Of course, if all visited states are stored or equivalently if no collision occurs, this algorithms performs an exhaustive verification. In this case, since the algorithm is a depth-first traversal, if the memory size is to small to store the complete graph, a replacement strategy can be used.

3.4 Testing for unboundedness of fifo channels

The depth–first traversal with replacement has also been proposed in [J9́1b, J9́1a] for the test of unboundedness of fifo channels in some specification models such as communicating finite state machines [Boc78], fifo–nets [MM81, FM85] and even Estelle programs [ISO89]. Unboundedness is generally undecidable [BZ83], but there exists a sufficient condition for unboundedness, which can be computed on the states of each transition sequence. Let S and S' be two states such that S' is reachable from S by the sequence of actions w. Let $C_j(S)$ and $C_j(S')$ be the contents of channel f_j in those states and $out_j(w)$ the projection of w on outputs in f_j. If variables (exept channel contents) in S and S' are identical and $\forall j, C_j(S).out_j(S) \leq C_j(S').out_j(w)$, then w can be infinitely fired from S' and reaches an infinite sequence of increasing states for the prefix ordering.

The reachability graphs we are working with are possibly infinite, so, even with a depth first traversal, we can only analyse finite sub–graphs. But the sufficient condition found on a finite sub–graph remains true on the underlying infinite graph.

Since the condition depends on transitions sequences, it can be computed during a depth–first traversal and is improved by storing and replacing some already visited states.

4 Conclusion and prospects

Dealing with the state space explosion problem, we have presented an alternative to the exhaustive construction of state graphs. The depth–first traversal insures an exhaustive traversal of all states and/or transitions of a reachability graph. It requires less memory since it theoretically only needs a memory large enough to store the longest acyclic sequence. In order to improve this technique, it is necessary to store some visited states. When the memory is full, visited states are randomly replaced by new states of the current sequence. We have shown that this method can significantly increase the size of the state graphs that can actually be analysed without excessively increasing the computation time.

As we saw, this method can be used for different kinds of verification. A few application examples have pointed out that it can certainly improve the verification tools in various domains such as bisimulation, Büchi acceptance, on–the–fly verification of temporal properties and test for unboundedness.

However, this technique does not solve all the problems. We still don't know the whole applicability domain of that method. For example, is it possible to verify branching time temporal logic properties with a depth–first traversal with replacement, and, if the answer is positive, is it efficient? We also know that this algorithm is not quite suited for all kinds of graphs. Perhaps an interesting problem would be to carefully study the structure of graphs for which it is well suited. We could then infer on the convenience of the method on some classes of transitions systems. Within a tool, the choose of the depth–first traversal in a particular verification could then be guided by the expected structure of graphs.

References

[BFH90] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer Aided Verification DIMACS 90*, " month="June, 1990.

[Boc78] G.V. Bochmann. Finite state description of communication protocols. *Computer Networks*, 2, October 1978.

[BZ83] D. Brand and P Zafiropulo. On communicating finite–state machines. *J.A.C.M*, 2:323–342, April 1983.

[CG87] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. 6^{th} *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada*, 1987.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Workshop on Computer Aided Verification, DIMACS 90*, " month="June, 1990.

[FM85] A. Finkel and G. Memmi. An introduction to fifo nets – monogeneous nets: a subclass of fifo nets. *Theoretical Computer Science*, 35:191–214, 1985.

[FM90] J.-C. Fernandez and L. Mounier. Verifying bisimulation on the fly. In *Third International Conference on Formal Description Techniques FORTE'90*, Madrid, November 1990.

[GS90] S. Graf and B. Steffen. Compositional minimization of finite state processes. In *Workshop on Computer Aided Verification DIMACS 90*, " month="June, 1990.

[GW91] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety. In *Computer Aided Verification, LNCS 575*, pages 332–342, Aalborg, Denmark., 1991.

[Hol85] G. Holzmann. Tracing protocols. *ATT Technical Journal*, 64(10):2413–2434, 1985.

[Hol87] G.J. Holzmann. Automated protocol validation in ARGOS, assertion proving and scatter searching. *IEEE trans. on Software Engineering, Vol 13, No 6*, June 1987.

[Hol89] G.J. Holzmann. Algorithms for automated protocol validation. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[ISO89] ISO. *Estelle: a Formal Description Technique based on an Extented State Transition Model*. ISO 9074 TC97/SC21/WG6.1, 1989.

[J91a] T. Jéron. Contribution à la validation des protocoles : test d'infinitude et vérification à la volée. Thèse, Université de Rennes I, Mai 1991.

[J91b] T. Jéron. Testing for unboundedness of fifo channels. In *STACS 91 : Symposium on Theoretical Aspects of Computer Science Hamburg, Germany*, february 1991. Springer–Verlag, LNCS #480, pages 322–333.

[JGM85] C. Jard, R. Groz, and J-F. Monin. Véda : a software simulator for the validation of protocol specifications. In *COMNET'85, Hungary*, October 1985.

[JGM88] C. Jard, R. Groz, and J.F. Monin. Development of VEDA: a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, volume 14,3, pages 339–352, March 1988.

[JJ89] C. Jard and T. Jéron. On-line model-checking for finite linear temporal logic specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989. Springer–Verlag, LNCS #407, pages 275–285.

[MM81] R. Martin and G. Memmi. Spécification et validation de systèmes temps réel à l'aide de réseaux de petri à files. Technical Report 3, Revue Tech. Thomson–CSF, Sept. 1981.

[PJ88] J.-M. Pageot and C. Jard. Experience in guiding simulation. *Protocol Specification, Testing and Verification, VIII, IFIP*, pages 207–218, June 1988.

[Pnu86] 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*, pages 510–584, 1986.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Workshop on Computer Aided Verification DIMACS 90*, " month="June, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata–theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science, Cambridge*, pages 322–331, June 1986.

[Wes78] C.H. West. General techniques for communication protocols. *IBM J. Res. Develop.*, 22, July 1978.

[Wes86] C.H. West. Protocol validation by random state exploration. In 6^{th} *IFIP International Workshop on Protocol Specification, Testing and Verification, Montréal, Gray rock.* North Holland, June 1986.