On-the-fly Verification of Finite Transition Systems [1]

Jean-Claude Fernandez,
Laurent Mounier

IMAG/LGI, BP53X, St Martin d'Hères
F-38041 Grenoble Cedex, FRANCE.
fernand@imag.fr

Claude Jard,
Thierry Jéron

IRISA, Campus de Beaulieu
F-35042 Rennes, FRANCE.
jard@irisa.fr

Abstract

The analysis of programs by the exhaustive inspection of reachable states in a finite state graph is a well-understood procedure. It is straightforwardly applicable to many description languages and is actually implemented in several industrial tools.

But 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. For numerous properties, it is not necessary to explicitly build this graph and an exhaustive depth–first traversal is often sufficient. This leads to an on–line algorithms for computing Büchi acceptance (in the deterministic case) and behavioral equivalences: they are presented in detail.

In order to avoid retraversing states, it is however important to store some of the already visited states in memory. To keep the memory size bounded (and avoid a performance falling down), visited states are randomly replaced. In most cases this depth–first traversal with replacement can push back significantly the limits of verification tools.

We call "on–the–fly verification", the use of algorithms based on a depth–first search (with replacement) of the finite state graph associated with the program to be verified.

Keywords : verification, Finite Transition Systems, model-checking, bisimulation, depth-first search.

[1] This work was partly funded by the french national project C^3 on parallelism.

Contents

1 Introduction

1.1 Motivation

Program verification is a branch of computer science whose purpose is "to prove programs correctness". Let us recall that correctness proofs 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.

Verification has been studied in theoretical computer science departments for a long time but it is rarely applied to real world problems. As a matter of fact, we must pay much more attention to practical problems such as the amount of space and time needed to perform verification.

A considerable need for such methods appeared these last ten years in different domains, such as design of asynchronous circuits, communication protocols 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 exist now 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. We refer to this technique as reachability analysis. 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 Limits of the reachability analysis

Reachability analysis is basically an exhaustive search yielding a rooted graph of global states. This technique is often called *perturbation* [31]. 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 at least 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. A profound study of algorithms dedicated to the reachability analysis has been conducted by G. Holzmann at the ATT Bell Labs since 1985 [15, 16, 17]. Let us recall some complexity results.

Let R be the number of reachable states. We can suppose that states are of constant size S. As we want to store and compare states, we can reasonably suppose that the memory is arranged as a balanced tree. The memory size M needed to store the state size is then at least $R.S$. Let $C(S)$ be the time needed for the comparison of two states. When the i^{th} state is generated for the first time, 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 in average 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

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

As an example, 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 in the order $T \simeq 6$ minutes.

In order to master the "state explosion", different complementary works have been conducted to reduce the size of the graph [5, 30, 3, 12, 13, 11]. 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 [32, 27, 21, 17].

1.3 State-of-the-art in on-the-fly verification

The 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 may be catastrophic, due to the re-generation of already visited states.

An intermediate method 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 called "verification on-the-fly".

It was first proposed in [15] in the context of partial verification as a possible method to restrict the state space during a "scatter" search. This idea was rediscovered in [22] and presently applied to complete verification by "on–line" model checking. Since then, similar ideas have been advocated in [6] and [9]. [6] presents efficient algorithms to verify properties given by Büchi automata and thus proposes a new solution to the verification of temporal properties on infinite behaviors of finite state programs ; their method consists of checking the emptyness of the automaton resulting from the product of the program and the property without explicitly constructing the strongly connected components of the automaton. [9] extends the technique to verify on-the-fly behavioral equivalences and preorders on transitions graphs. The core of the method is to traverse (during its generation) a kind of synchronous product of finite transitions systems. In [10, 23] new on–the–fly verification algorithms have been designed, prototyped and measured. This paper presents these algorithms in a uniform manner and will serve as a basis for integration in a verification tool (namely the CAESAR/ALDÉBARAN tool [11]).

1.4 Paper organisation

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. They form the on–the–fly kernel of the verification tool. The second part of the paper shows how different verification problems can be solved as an exhaustive traversal of a "product" transition system. The on–the–fly verification algorithms are given in detail using the kernel, seen

as a simulator with "holes". We present four algorithms:

- verification of acceptance of a finite transition system by a deterministic Büchi automaton (safety properties),
- verification of bisimulation equivalence in the deterministic and non-deterministic cases,
- testing of the unboundedness of Fifo channels.

We conclude with some prospects.

2 The on-the-fly kernel: basic traversal algorithms

We saw above that the main drawback of the perturbation technique is the memory size needed to perform the graph generation for real life systems. Now, there are some verification problems for which a traversal of all states and transitions is sufficient. 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 belong 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 of

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

However, a depth–first traversal can be significantly improved if the whole available memory is used [19]. Actually, since $D_{max}.S < M$ (where M is the size of the memory), one can use the remainder of the memory to store already visited states and consequently to avoid re-generation of some states. We present this technique and show by means of examples that it can be efficiently used to analyse real size graphs which are too large to fit in memory.

2.1 The algorithm

```
procedure DFSR (S0: state; var N, V, P: set_of_states;
        Act, Act_Stack, Act_NPV, Act_Pop: procedure;
        Cond_Null: function;
        var res: result
        );
var
    St : stack; (* -- states of the current sequence -- *)
    St_Trans : stack; (* -- stack of transitions of the current sequence -- *)
    St_Ens_Trans : stack; (* -- stack of sets of pending transitions -- *)
    S, S' : state; t : transition;
begin
    St := nil; St_Trans := nil; St_Ens_Trans := nil;
    push (S0, St); push (firable(S0), St_Ens_Trans);
    while St ≠ nil do begin
        S := top (St); (* -- current state -- *)
        if top (St_Ens_Trans) ≠ ∅ then begin
            t := extract_one_of (top(St_Ens_Trans));(* -- choose and remove -- *)
            push ( t, St_Trans);
            S' := succ (S,t);
            [Act (S', St, St_Trans, res);]
            case search (S') of
                Null : begin
                    if memory_full then (* -- replacement -- *)
                        if V = ∅ then res := memory_overflow
                        else V := V - { one_of(V) };
                    if [Cond_Null (S')] then N := N ∪ {S'}
                    else begin
                        push (S', St);
                        push (firable(S'), St_Ens_Trans);
                        end;
                    end
                Stack : [Act_Stack (S, S', res);]
                Nec, Perm, Vis: [Act_NPV (S, S', res);];
                end;
            pop(St_Trans);
            end
        else begin (* -- top(St_Ens_Trans) = ∅ -- *)
            pop (St); pop(St_Ens_Trans)
            V := V ∪ {S};
            [Act_Pop (S, res);]
            end;
    end;
end;
```

The algorithm performing a depth–first traversal with replacement is described above. The algorithm is described as an exhaustive simulation with

holes. The contents of the holes depend on the kind of verification used; they describe the conditions (*Cond_Null*) and actions (*Act, Act_Stack, Act_NPV* and *Act_Pop*) that must be performed during the search.

The parameters of the algorithm are the initial state S_0 of the current DFSR, i.e. Depth First Search with Replacement, a set V of already visited states that can be replaced, two sets N and P of states which cannot be replaced, the four procedures *Act, Act_Stack, Act_NPV, Act_Pop*, the boolean function *Cond_Null*, and a result *res*.

One DFSR uses three stacks *St*, *St_trans* and *St_Ens_Trans* which respectively contain the states, transitions and pending transitions (those which are not yet fired) of the current sequence. We also assume the existence of an implicit memory of size M composed of the states belonging to *St*, N, V, P and we always insure that these sets are disjoint.

The DFSR algorithm uses several primitive functions and procedures such as the classical *push*, *pop* and *top* which operate on stacks, *firable* which gives the set of firable transitions from a state, *one_of* which chooses an element in a set, *extract_one_of* which chooses and removes an element from a set, *succ* which gives the successor of a state after firing some transition, *search* which searches a state in the memory. Its result is *Null* if the state is not in the memory and either *Stack*, *Nec*, *Perm* or *Vis* if it respectively belongs to *St*, N, P or V.

```
procedure DFS_Simple;
var
    S0 : state;
    N , V, P : set_of_states;
    res : result;

begin
    V := ∅; N := ∅; P := ∅
    S0 := initial_state;
    DFSR (S0, N, V, P, nop, nop, nop, nop, false, res);
end;
```

Let us explain the behavior of the algorithm in the case of a simple DFS, i.e. Depth First Search, in which the actions *Act, Act_Stack, Act_NPV* are the null operation *nop* and *Cond_Null* is the constant boolean function *false* (this implies that N is always empty).

Initially, St contains S_0 and St_Ens_Trans contains the set of firable transitions from S_0. The algorithm is then a loop which stops as soon as the stack St is empty i.e. when all states which are accessible from S_0 have been visited. or when a memory overflow is detected. It differs from a classical depth first search by the replacement which possibly happens when a newly generated state S' does not belong to the memory. In this case, we must push S' in St. But the memory (composed of St, $N = \emptyset$, $P = \emptyset$ and V) may be full. In this case, either V is empty and the algorithm fails be memory overflow or we can remove one state from V and then push S' in St.

The simple DFS algorithm with replacement can be used on any graph such that $D_{max}.S \leq M$. But, contrarily to a classical traversal, this is not a necessary condition for the termination because states of the longest acyclic sequences may be reached by shorter 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 S to S' 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 the evaluation of the transitions.

2.2 Time complexity of a simple DFS with replacement

Note that we always have $(|St| + |V|).S \leq M$ and the boolean variable $memory\_full$ means $(|St| + |V|).S = M$ and is a stable property. Let R_{ins} be the number of insertions of states in the memory i.e. $St \cup V$. 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 exactly once, so $R_{ins} = R$. The time complexity is then approximatively the same.

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 $St \cup V$ 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 $St \cup V$, we must remove some state S_{del} from

V before pushing S' into St. The way this replacement is performed influences the total number of generated states R_{gen}.

We also suppose that the whole memory $St \cup V$ (or $St \cup V \cup N$ if $N \neq \emptyset$) 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 approximatively

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

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 almost 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 $R.S$.

In practice, we hope that R_{ins} remains close to R.

The choice of a replacement strategy is then essential in such an algorithm. Several strategies have been looked at. As already noticed in [16], the best one seems to be random replacement, as in general nothing about the structure of the graph can be known in advance. It is easily performed and has no performance drop for particular graphs.

2.3 Experiments

The depth–first traversal with replacement has been used for different kinds of graphs: accessibility graphs of communication protocols modelled by communicating finite state machines, and 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 $St \cup V$ and the execution time. If $M = M_{max} = R.S$ then $R_{ins} = R$. As M decreases, R_{ins} increases. But it increases very slowly until M comes close to M_{min}. R_{ins} is then less than twice R. Finally, R_{ins} explodes but the memory is significantly reduced compared to pertubation before explosion. The execution time T has a similar form. For this example, with a memory size of 40% of $R.S$ we have only 70% more states insertion, which results in a time increase of 50%.

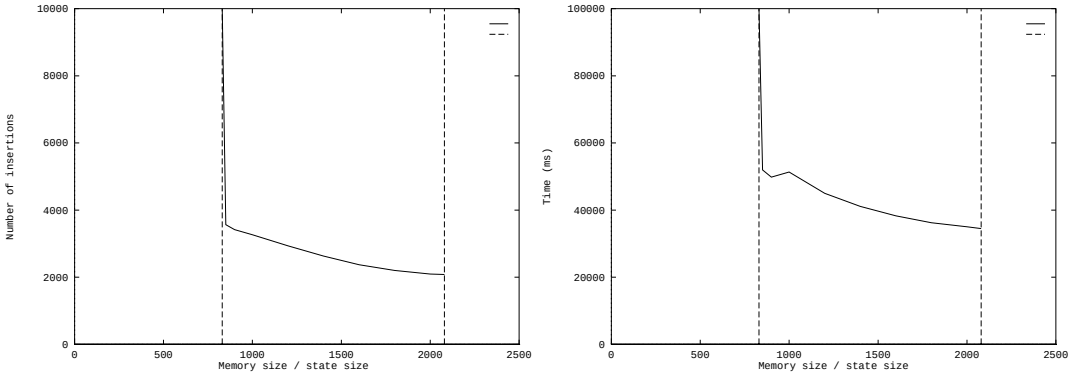

Figure 1: Number of generated states and execution time

Many examples have been tested with this traversal. They almost always gave the same type of curves. But we can only decrease the memory size down to a value between $D_{max}.S$ and $G_{max}.S$. Thus when D_{max} is small with respect to R, one can reduce M 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 with D_{max} 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 small and R_{ins} and T increase very quickly.

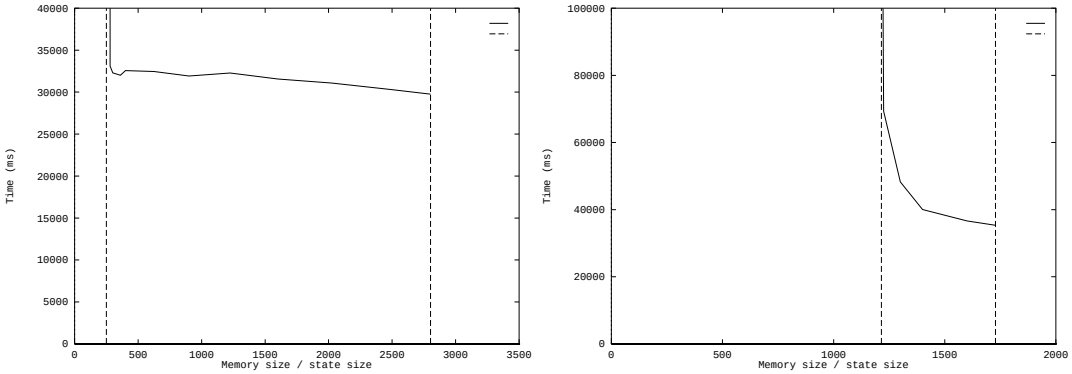

Figure 2: Execution time in extremal cases

3 Product system analysis

3.1 Behavioral equivalences and preorders

One of the successful approaches used for the verification of communicating processes consists in comparing different specifications of a given system by means of behavioral equivalence and preorder relations. More precisely, if $Spec_1$ denote the more abstract specification of the system and $Spec_2$ the more detailed one, it is possible to check whether $Spec_2$ agrees with $Spec_1$: Let R be an appropriate equivalence relation or preorder relation. Then $Spec_2$ agrees with $Spec_1$ if and only if $Spec_1$ R $Spec_2$. With $Spec_i$ a labeled transition system $\mathcal{S}_i$ (LTS for short) is associated and R is an equivalence relation or preorder relation on LTS.

Bisimulation equivalences and simulations equivalences or preorders play a central role in the verification of communicating systems. Many efficient algorithms for computing various bisimulation equivalences (strong, weak, branching) were proposed [24, 2, 28, 8, 25, 29, 14, 10, 3]. According to the definition of an equivalence relation which is either a set of state classes or a binary relation on the state space, the methods consists of refining a current partition until each class is stable or checking if a pair of states belonging to the current relation are bisimilar. In [10], we have shown that it is sufficient to define a particular synchronous product between two LTS, parametrized by a simulation or a bisimulation.

Recall that a LTS $\mathcal{S}$ is a rooted state graph with a labelling function

$< Q, A, T, q_0 >$ where Q is a finite set of states, A a finite set of actions, $T \subseteq Q \times A \times Q$ the transition relation, and q_{0_1} the initial state. We use also the notation $p \xrightarrow{a}_T q$ for $(p, a, q) \in T$.

Let $\mathcal{S}_i =< Q_i, A, T_i, q_{0\,i} >$ be two LTS. We recall the definition of simulation and bisimulation. Let $\lambda \subseteq A^*$, and let $p, q \in Q$. We write $p \xrightarrow{\lambda}_T q$ iff: $\exists u_1 \cdots u_n \in \lambda$ and $\exists q_1, \cdots, q_{n-1} \in Q$ and $p \xrightarrow{u_1}_T q_1 \xrightarrow{u_2}_T q_2 \cdots q_i \xrightarrow{u_{i+1}}_T q_{i+1} \cdots q_{n-1} \xrightarrow{u_n}_T q$. Let Π be a family of disjoint languages on A.
$\mathcal{A}\mathrm{ct}_\Pi(q, T) = \{\lambda \in \Pi \mid \exists q' \,.\, q \xrightarrow{\lambda}_T q'\}$.

Definition 3.1 *(simulation) Let Π be a family of disjoint languages on A. We define inductively a family of simulations R_k^Π by:*

$$\begin{aligned} R_0^\Pi &= Q_1 \times Q_2 \\ R_{k+1}^\Pi &= \{(p_1, p_2) \mid \forall \lambda \in \Pi \,. \\ &\qquad \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\Rightarrow\; \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\wedge\; (q_1, q_2) \in R_k^\Pi))\} \end{aligned}$$

The simulation preorder for Π is $\sqsubseteq^\Pi = \bigcap_{k=0}^{\infty} R_k^\Pi$, the simulation equivalence is $\approx^\Pi = \bigcap_{k=0}^{\infty} (R_k^\Pi \cap R_k^{\Pi^{-1}})$.

Definition 3.2 *(bisimulation) Let Π be a family of disjoint languages on A. We define inductively a family of bisimulations R_k^Π by:*

$$\begin{aligned} R_0^\Pi &= Q_1 \times Q_2 \\ R_{k+1}^\Pi &= \{(p_1, p_2) \mid \forall \lambda \in \Pi \,. \\ &\qquad \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\Rightarrow\; \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\wedge\; (q_1, q_2) \in R_k^\Pi)) \\ &\qquad \forall q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\Rightarrow\; \exists q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\wedge\; (q_1, q_2) \in R_k^\Pi))\} \end{aligned}$$

From these general definitions, several simulation and bisimulation relations can be defined. The choice of a class Π corresponds to the choice of an *abstraction criterion* on the actions. The *strong simulation* and the *strong bisimulation* are defined by $\Pi = \{\{a\} \mid a \in A\}$, the *w-bisimulation* is the bisimulation equivalence defined by $\Pi = \{\tau^* a \mid a \in A \;\wedge\; a \neq \tau\}$, the *safety preorder* is the simulation preorder defined by $\Pi = \{\tau^* a \mid a \in A \;\wedge\; a \neq \tau\}$ and the *safety equivalence* is the simulation equivalence where $\Pi = \{\tau^* a \mid a \in A \;\wedge\; a \neq \tau\}$.

We define the product $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ between the two LTS $\mathcal{S}_1$ and $\mathcal{S}_2$, and then we show how the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ are related by R^Π can be expressed as a simple criterion on the execution sequences of this product. We use p_i, q_i, p_i', q_i' to range over Q_i. We use R^Π and R_k^Π to denote either simulations or bisimulations ($R^\Pi = \bigcap_{k=0}^{\infty} R_k^\Pi$).

The LTS $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ is defined as a synchronous product of $\mathcal{S}_1$ and $\mathcal{S}_2$: a state (q_1, q_2) of $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ can perform a transition labeled by λ if and only if the state q_1 (belonging to $\mathcal{S}_1$) **and** the state q_2 (belonging to $\mathcal{S}_2$) can perform a transition labeled by λ. Otherwise,

- in the case of a simulation, if **only** the state q_1 can perform a transition labeled by λ, then the product has a transition from (q_1, q_2) to the sink state noted *fail*.
- in the case of a bisimulation, if **only one** of the two states (q_1 or q_2) can perform a transition labeled by λ, then the product has a transition from (q_1, q_2) to the sink state *fail*.

Definition 3.3 *We define the LTS* $\mathcal{S} = \mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ *by:*
$S = (Q, A, T, (q_{0\,1}, q_{0\,2}))$, *with* $Q \subseteq (Q_1 \times Q_2) \cup \{fail\}$, $A = (A_1 \cap A_2) \cup \{\phi\}$, *and* $T \subseteq Q \times A \times Q$, *where* $\phi \notin (A_1 \cup A_2)$ *and* $fail \notin (Q_1 \cup Q_2)$.
T *and* Q *are defined as the smallest sets obtained by the applications of the following rules:* $R0$, $R'1$ *and* $R'2$ *in the case of a simulation,* $R0$, $R1$ *and* $R2$ *in the case of a bisimulation.*

$$(q_{01}, q_{02}) \in Q \qquad [R0]$$

$$\frac{(q_1, q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Pi(q_1) = \mathcal{A}\mathrm{ct}_\Pi(q_2),\ \lambda \in \Pi,\ q_1 \xrightarrow{\lambda}_{T_1} q_1'\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{\lambda}_T (q_1', q_2')\} \in T} \qquad [R1]$$

$$\frac{(q_1, q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Pi(q_1) \subseteq \mathcal{A}\mathrm{ct}_\Pi(q_2),\ \lambda \in \Pi,\ q_1 \xrightarrow{\lambda}_{T_1} q_1'\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{\lambda}_T (q_1', q_2')\} \in T} \qquad [R'1]$$

$$\frac{(q_1, q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Pi(q_1) \neq \mathcal{A}\mathrm{ct}_\Pi(q_2),}{\{fail\} \in Q, \{(q_1, q_2) \xrightarrow{\phi}_T fail\} \in T} \qquad [R2]$$

$$\frac{(q_1,q_2)\in Q,\ \mathcal{A}\mathrm{ct}_\Pi(q_1)\not\subseteq\mathcal{A}\mathrm{ct}_\Pi(q_2),}{\{fail\}\in Q,\{(q_1,q_2)\xrightarrow{\phi}_T fail\}\in T}\qquad[R'2]$$

Note that $(p_1,p_2)\xrightarrow{\phi}_T fail$ if and only if $(p_1,p_2)\notin R_1^\Pi$.

The following proposition gives the promised cartesian product on the execution sequences of $\mathcal{S}$, allowing to decide that $\mathcal{S}_1$ and $\mathcal{S}_2$ are not related by R^Π in terms of the execution sequences of $\mathcal{S}_1\times_{R^\Pi}\mathcal{S}_2$.

Proposition 3.1 *$(q_{01},q_{02})\notin R^\Pi$ if and only if it exists an elementary execution sequence σ of S such that:*

- $\sigma=\{(q_{01},q_{02})=(p_0,q_0),(p_1,q_1),\ \ldots\ (p_k,q_k),fail\}$.
- $\forall i\ .\ 0\le i\le k,\ (p_i,q_i)\notin R^\Pi_{k-i+1}$.

If one of the two LTS is deterministic, proposition 3.1 can be improved. For a state (q_1,q_2) of $\mathcal{S}_1\times_{R^\Pi}\mathcal{S}_2$, $(q_1,q_2)\in R_k^\Pi$ if and only if *fail* is not a successor of (q_1,q_2) and all the successors (q_1',q_2') of (q_1,q_2) verify $(q_1,q_2)\in R^\Pi_{k-1}$.

Proposition 3.2 *Let us suppose that S_2 is deterministic (or S_1 if the $(R_k^\Pi)_{k\ge0}$ are bisimulations). Then:*
$S_1\not R^\Pi S_2\Leftrightarrow\exists\sigma\in Ex(q_{01},q_{02})\ .\ \exists k>0\ .\ \sigma(k)=\text{fail}.$

According to this proposition, if at least one of the two LTS S_1 or S_2 (resp. S_2) is deterministic then S_1 and S_2 are not bisimilar (resp. similar) if and only if it exists an execution sequence of $\mathcal{S}_1\times_{R^\Pi}\mathcal{S}_2$ containing the state *fail*.

3.2 On–line model checking

Let $\mathcal{S}_1=<Q_1,A,T_1,q_{01}>$ be the labeled transition system associated with the specification *Spec*.

Suppose that a property $\mathcal{P}$ can be expressed by a deterministic Büchi automaton $\mathcal{B}=<Q_2,A,T_2,q_{02},F_2>$ 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_{02} 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 $\mathcal{B}$: $q_{02}\xrightarrow{a_1}_{T_2}q_1\ldots q_{n-1}\xrightarrow{a_n}_{T_2}q_n\ldots$ 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 $\mathcal{B}$.

In the case that the Büchi automaton may be non 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 show that there is a very simple algorithm which performs this verification without complementation and without computation of strongly connected components.

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.

Definition 3.4 *The synchronous product* $\mathcal{S} =< Q, A, T, q_0, F >$ *of* $\mathcal{S}_1$ *and* $\mathcal{B}$ *is defined by:*

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

$$\frac{q_1, q_1' \in Q_1; q_2, q_2' \in Q_2,\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{(q_1, q_2) \xrightarrow{a}_{T} (q_1', q_2')}$$

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 reachable cycle of the graph contains a vertex 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 corresponding edges) is acyclic. And $\mathcal{S}'$ is acyclic if and only if a depth-first traversal of $\mathcal{S}'$ doesn't detect any cycle.

4 On-the-fly verification

4.1 Büchi acceptance for deterministic case

As we saw above, the problem is to detect whether the subgraph $\mathcal{S}'$ is acyclic. But we don't want to first build $\mathcal{S}$ and then remove vertices of F. We would like to check whether $\mathcal{S}'$ is acyclic during a traversal of $\mathcal{S}$. The subgraph $\mathcal{S}'$ is not necessarily weakly connected. But each weakly connected component of $\mathcal{S}'$ is reachable in $\mathcal{S}$ from a state in F or from q_{init}. And states in F are reachable in $\mathcal{S}$ from q_{init}.

The algorithm that we propose is a particular traversal of $\mathcal{S}$ consisting in several partial DFS. Each partial DFS is rooted by a state in F or q_{init} and explores every state accessible in $\mathcal{S}'$ from the actual root. Thus we cannot go beyond states of F but we discover all of them during the partial traversals.

If the memory is large enough to store the whole state graph $\mathcal{S}$, the algorithm terminates and detects a loop in $\mathcal{S}'$ if and only if one exists. A loop in $\mathcal{S}'$ is detected if **search**(S') gives the result **Stack** and $S' \notin F$ (see the action **Act_Stack**). Furthermore, the algorithm is linear in the size of $\mathcal{S}$ as every edge of $\mathcal{S}$ is traversed once and only once. It is then more efficient than a classical Tarjan's algorithm which calculates the strongly connected components of $\mathcal{S}$ and detects if one of them contains a state in F.

Now if the memory is too small, we can use the replacement strategy. The algorithm is ensured to terminate correctly if loops are detected in $\mathcal{S}'$ and every state from $F \cup \{q_0\}$ initiates one and only one partial DFSR. Thus we can remove every state from V which does not belong to $F \cup \{q_0\}$.

In order to perform the algorithm on the basis of our partial DFS, we need to fill some of the holes. We first need a set N which contains the roots of the depth-first traversals not yet performed i.e. the states of F which have already been discovered but not used. And we need a set P containing the roots of preceding partial DFSR.

If a new state $q \in F$ is reached, it is added to N and successors of q are not explored in the present DFSR (they will be explored in the traversal initiated in q). When the DFSR starting in the root q_{init} is finished, q_{init} is added to the set P in *Act_Pop*, in such a way that every visited terminal state is either in N or in P. If a cycle is detected in $q \notin F$ this simply signifies that a cycle of $\mathcal{S}'$ is detected.

The algorithm stops when N is empty and $Spec \models \mathcal{P}$ if and only if no

cycle of $\mathcal{S}'$ is detected. This algorithm, in which the actions in which the actions *Act, Act_NPV* are the null operation *nop*, is described below:

```
procedure DFS_Buchi;
var
    q_init : state;
    N, V, P: set_of_states;
    res : result;
function Cond_Null (q' : state) : boolean;
    begin Cond_Null := (q' ∈ F) end;

procedure Act_Stack(q, q' : state; var res : result);
    begin if (q' ∉ F) then res:=error; end;

procedure Act_Pop(q : state; var res : result);
    begin
        if q = q_init then begin
            (* -- initial states of each DFSR must be preserved from replacement -- *)
            V := V - {q};
            P := P ∪ {q};
        end;
    end

begin
    V := ∅; N := ∅;
    q_init := initial_state; (* -- initial_state is q0 = (q0_1, q0_2) -- *)
    repeat
        DFSR (q_init, N, V, P, nop, Act_Stack, nop, Act_Pop, Cond_Null, res);
        if N ≠ ∅ then
            q_init := extract_one_of(N);
    until (N = ∅ or res = error or res = memory_overflow);
end;
```

4.2 Bisimulation

In the previous section, we have expressed the bisimulation and the simulation between two LTS S_1 and S_2 in terms of the existence of a particular execution sequence of their product $\mathcal{S}_1 \times_{R^{\Pi}} \mathcal{S}_2$. Now we show that this verification can be realized by performing depth-first searches (DFS for short) on the LTS $\mathcal{S}_1 \times_{R^{\Pi}} \mathcal{S}_2$. Consequently, the algorithm does not require to construct the two LTS previously : the states of $\mathcal{S}_1 \times_{R^{\Pi}} \mathcal{S}_2$ are generated during the DFS ("on the fly" verification), but not necessarily all stored. And the most important is that the transitions do not have to be stored.

We note n_1 (resp. n_2) the number of states of S_1 (resp. S_2), and n the number of states of $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ ($n \leq n_1 \times n_2$). We describe the algorithm considering the two following cases:

Deterministic case: if R^Π represents a simulation (resp. a bisimulation) and if S_2 (resp. either S_1 or S_2) is deterministic, then, according to proposition 3.2, it is sufficient to check whether or not the state *fail* belongs to $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$, which can be easily done by performing a usual DFS of $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$. The verification is then reduced to a simple reachability problem in this graph. Consequently, if we store all the visited states during the DFS, the time and memory complexities of this decision procedure are $O(n)$.

General case: in the general case, according to the proposition 3.1, we have to check the existence of an execution sequence σ of $\mathcal{S}_1 \times_{R^\Pi} \mathcal{S}_2$ which contains the state *fail* and which is such that for all states (q_1, q_2) of σ, $(q_1, q_2) \notin R_k^\Pi$ for a certain k. According to the definition of R_k^Π, this verification can be done during a DFS as well if:

- the relation R_1^Π can be checked.
- for each visited state (q_1, q_2), the result $(q_1, q_2) \in R_k^\Pi$ is synthesized for its predecessors in the current sequence (the states are then analyzed during the back tracking phase).

More precisely, the principle of the general case algorithm is the following: if R^Π is a simulation (resp. a bisimulation) we associate with each state (q_1, q_2) a set $Equiv\_List(q_1, q_2)$ of size $|T_1[q_1]|$ (resp. $|T_1[q_1]| + |T_2[q_2]|$). During the analysis of each successor (q_1', q_2') of (q_1, q_2), whenever it happens that $(q_1', q_2') \in R^\Pi$ then q_1' is inserted into $Equiv\_List(q_1, q_2)$ (resp. q_1' and q_2' are inserted into $Equiv\_List(q_1, q_2)$). Thus, when all the successors of (q_1, q_2) have been analyzed, $(q_1, q_2) \in R^\Pi$ if and only if $Equiv\_List(q_1, q_2) = T_1[q_1]$ if R^Π is a simulation (resp. $Equiv\_List(q_1, q_2) = T_1[q_1] \cup T_2[q_2]$ if R^Π is a bisimulation).

As in the deterministic case algorithm, to reduce the time complexity of the DFS the usual method would consist in storing all the visited states (including those which do not belong to the current sequence) together with the result of their analysis (i.e, if they belong or not to R^Π). Unfortunately, this solution cannot be straightly applied:

During the DFS, the states are analyzed in a postfixed order. Consequently, it is possible to reach a state which has already been visited, but not yet analyzed (since the visits are performed in a prefixed order). Therefore, the result of the analysis of such a state is unknown (it is not available yet). We propose the following solution for this problem: The result returned by the function DFSR may be TRUE, FALSE or UNRELIABLE. The algorithm then consists in a sequence of calls of $DFSR$ (each call increasing the set Non_equiv_States), until the result belongs to $\{TRUE, FALSE\}$.

We call the *status* of a state the result of the analysis of this state by the function DFSR. The status of (q_1, q_2) is "$\sim$" if $(q_1, q_2) \in R^{\Pi}$, and is "$\not\sim$" otherwise.

If R^{Π} is a simulation then

$$Equiv\_List(p,q) \;=\; \{p' \mid \exists q' \,.\, (p',q') \in firable(p,q) \;\wedge\; status(p',q') = \sim\}$$

If R^{Π} is a bisimulation then

$$\begin{aligned} Equiv\_List(p,q) \;&=\; \{p' \mid \exists q' \,.\, (p',q') \in firable(p,q) \;\wedge\; status(p',q') = \sim\} \\ &\cup\; \{q' \mid \exists p' \,.\, (p',q') \in firable(p,q) \;\wedge\; status(p',q') = \sim\} \end{aligned}$$

Whenever a state already visited but not yet analyzed (i.e, which belongs to the stack) is reached, then we assume its status to be "$\sim$". If, when the analysis of this state completes (i.e, when it is popped), the obtained status is "$\not\sim$", then a TRUE answer from the DFSR is not reliable, the result returned is UNRELIABLE (a wrong assumption was used), and another DFS has to be performed. On the other hand, a FALSE answer is always reliable.

We need a set *Scc_Roots* in order to store the roots of the strongly connected components encountered during the exploration.

The algorithm, in which the action Act is the null operation nop, dealing with the bisimulation relation is the following:

```
procedure DFS_Bisimu
var
    S_0 : state;
    Non_equiv_States, P, Visited, Scc_Roots : set_of_states;
    Equiv_List : set of set of states;
    res : result;
function Cond_Null (S' : state);
    begin Cond_Null:= (fail ∈ successors(S')); end;

procedure Act_Stack (S, S' : state; var res : result)
    begin
        Scc_Roots := Scc_Roots ∪ {S'};
        Equiv_List(S) := Equiv_List(S) ∪ {q'_1} ∪ {q'_2} (* -- S' = (q'_1, q'_2)  -- *); (1)
    end;

procedure Act_NPV (S, S' : state; var res : result)
    begin
        if S ∉ Non_equiv_States then
            Equiv_List(S) := Equiv_List(S) ∪ {S'}; (1)
        end;
    end;

procedure Act_Pop ( S : state; var res : result)
    begin
        if Equiv_List(S) = T_1[q_1] ∪ T_2[q_2] (*  -- S = (q_1, q_2)  -- *) then (2)
            Equiv_List(top (St)) := Equiv_List(top (St)) ∪ {S}; (3)
        else begin
            V := V - {S};
            N := N ∪ {S};
            if S ∈ Scc_Roots then res := unreliable;
        end;
    end;

begin
    Non_equiv_States := ∅;
    S_0 := (q_{0_1} , q_{0_2});
    repeat
        Scc_Roots :=∅;
        Visited :=∅;
        DFSR (S_0, Non_equiv_States, Visited, P, nop, Act_Stack, Act_NPV, Act_Pop  Cond_Null, res);
        res := Equiv_List (S_0) = T_1[q_{0_1}] ∪ T_{0_2}[q_2];(4)
    until result ∈ {true, false, memory_overflow}
return result
end;
```

The algorithm dealing with the simulation is straightly obtained by replacing:

(1) $\text{Equiv\_List}((q_1, q_2)) := \text{Equiv\_List}((q_1, q_2)) \cup \{q'_1\}$

(2) $\text{Equiv\_List}((q_1, q_2)) = T_1[q_1]$

(3) Equiv_List(top(St)) := Equiv_List(top(St)) $\cup \{q_1\}$

(4) Equiv_List((q_{0_1}, q_{0_2})) = $T_1[q_1]$

The algorithm terminates, and it returns TRUE if and only if the two LTS are bisimilars.

The time requirement for the function $DFSR$ is $O(n)$. In the worst case, the number of calls of this function may be n. Consequently, the theoretical time requirement for this algorithm is $O(n^2)$. In practice, it turns out that only 1 or 2 DFS are required to obtain a reliable result. Moreover, whenever the LTS are not bisimilar, the time requirement is always $O(n)$.

4.3 Testing for unboundedness of fifo channels

The depth–first traversal with replacement has also been proposed in [20, 19] for the test of unboundedness of fifo channels in some specification models such as communicating finite state machines [1], fifo–nets [26, 7] and even Estelle programs [18]. Unboundedness is generally undecidable [4], 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 (except channel contents) in S and S' are identical and $\forall j, C_j(S).out_j(w) \leq C_j(S').out_j(w)$, 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. The algorithm, in which the actions *Act_Stack, Act_NPV, Act_pop* are the null operation *nop* and *Cond_Null* is the constant boolean function *false*, is described below:

```
procedure DFS_Unbound;
var
    S0 : state;
    N , V, P : set_of_states;
    res : result;
procedure Act ( S' : state; St, St_Trans : stack; var res : resultat);
    var
        S : state;
        unb : boolean;
        w : transitions_sequence;
    begin
        unb := false;
        S := top (St);
        repeat
            w := seq_from_to (S, S');
            unb := ∀j, Cj(S).outj(w) ≤ Cj(S').outj(w);
            S := pred (S);
        until unb or (S = S0);
        if unb then res := unbounded;
    end

begin
    V := ∅; N := ∅; P := ∅;
    S0 := initial_state;
    DFSR (S0, N, V, P, Act, nop, nop, false, res);
end;
```

5 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.

We have explicitly given those new algorithms. After their prototyping, we are implementing them in the verification workstation called Open-Caesar.

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

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

[2] T. Bolognesi and S.A. Smolka. Fundamental results for the verification of observational equivalence. In H.Rudin and C.H. West, editors, *Protocol Specification, Testing and Verification VII*, 1987.

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

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

[5] 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.

[6] 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*, June 1990.

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

[8] J. C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.

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

[10] J.-C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In *CAV 91: Symposium on Computer Aided Verification*, AAlborg, Denmark, June 1991.

[11] Hubert Garavel and Joseph Sifakis. Compilation and verification of lotos specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa)*, IFIP, North-Holland, Amsterdam, June 1990.

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

[13] P. Godefroid and P. Wolper. A partial approach to model checking. In 6^{th} *IEEE Symposium on Logic in Computer Science, Amsterdam*, July 1991.

[14] Jan Friso Groote and Frits Vaandrager. *An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence*. CS-R 9001, Centrum voor Wiskunde en Informatica, Amsterdam, January 1990.

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

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

[17] 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.

[18] ISO 9074. *Estelle: a Formal Description Technique based on an Extented State Transition Model*. ISO TC97/SC21/WG6.1, 1986.

[19] T. Jéron. Contribution à la validation des protocoles : test d'infinitude et vérification à la volée. Thèse de doctorat d'informatique de l'Université de Rennes 1, Mai 1991.

[20] 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.

[21] C. Jard, R. Groz, and J.F. Monin. Development of VEDA: a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, March 1988.

[22] 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.

[23] C. Jard and T. Jéron. Bounded memory algorithm for verification on-the-fly. In *CAV 91: Symposium on Computer Aided Verification*, AAlborg, Denmark, June 1991. Also available as INRIA Research Report n^o 11462.

[24] P. Kanellakis and S. Smolka. Ccs expressions, finite state processes and three problems of equivalence. In *Proceedings ACM Symp. on Principles of Distribued Computing*, 1983.

[25] K.G. Larsen. *Context–Dependent Bisimulation Between Processes.* Technical Report CST-37-86, Departement of Computer Science, University of Edimburgh, May 1986. Ph.D.

[26] 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.

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

[28] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput., No. 6*, 16, 1987.

[29] Huajun Qin. Efficient verification of determinate proce sses. In J.C.M Baeten and J.F. Groote, editors, *Proceedings of the 2nd International Confer ence on Concurrency Theory (CONCUR'91)*, pages 470–479, North-Holland, Amsterdam, August 1991.

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

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

[32] 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.