# Property Driven Distribution of Nested DFS*

Jiří Barnat, Luboš Brim, and Ivana Černá

Faculty of Informatics, Masaryk University Brno,
Botanická 68a, Brno, Czech Republic
{barnat,brim,cerna}@fi.muni.cz

**Abstract.** In this paper we propose a distributed algorithm for model-checking LTL formulas that works on a network of workstations and effectively uses the decomposition of the formula automaton to strongly connected components to achieve more efficient distribution of the verification problem. In particular, we explore the possibility of performing a distributed nested depth-first search algorithm.

## 1 Introduction

Many of the logic programming systems (e.g. concurrent constraint, multi-agent) can be semantically modelled as labelled transition systems. On top to the logic programming inherent approach to the correctness, this opens a possibility to provide additional validity arguments by using techniques developed for verification of transition systems, like equivalence checking or model checking. Moreover, for systems that involve synchrony, e.g. [8, 4], is model checking an appropriate and well developed technique.

In computational logic approach the system description in some logic formalism is at the heart of the paradigm. Therefore, it is at the hand to employ the information gained from the formulas to model check the system. The main purpose of the paper is to propose a particular technique that uses information gained from the model of a formula to make the verification more efficient. In some cases it is even possible to extract the information directly from the formula itself.

As with all complex concurrent systems (either hardware or software), techniques to avoid the state-explosion problem are required [7]. Many different methods for alleviating the state-explosion problem have been proposed and incorporated into verification tools. Recently, several attempts to use multiprocessors and networks of workstations have been undertaken.

One of the main issues in distributed *explicit state* model checking algorithms is how to partition the state space [1, 6, 11] among the individual computers called here *network nodes*. Some of the state space generation and partition techniques exploit certain characteristics of the system, and hence work well for systems possessing these characteristics, but fail to work well for systems which do not have them. In some cases it is possible to decide in advance whether the system under consideration has the required characteristic. However, in most situations this is not the case.

Recently, there have been proposed two approaches that use additional information about the state space to make the verification process more efficient.

* This work has been supported in part by the Grant Agency of Czech Republic grant No. 201/00/1023.

```
proc DFS(state)
  if (state, 0) ∉ visited
    then visited := visited ∪ {(state, 0)}
         in_stack := in_stack ∪ {state}
         foreach s ∈ Succ(state) do
           DFS(s)
         od;
         if Accepting(state)
           then NDFS(state)
         fi
         in_stack := in_stack \ {state}
  fi
end
```

```
proc NDFS(state)
  if (state, 1) ∉ visited
    then visited := visited ∪ {(state, 1)}
         foreach s ∈ Succ(state) do
           if s ∈ in_stack
             then Report("Cycle")
             else
                  NDFS(s)
           fi
         od;
  fi
end
```

**Fig. 1.** Nested DFS Algorithm

In [9] and in [3] the authors have exploited the particular structure of the verified property.

In this paper we propose a technique that uses the verified property to partition the state space in a distributed on-the-fly automata-based model checking of LTL properties. In automata-based LTL model checking [13] the verification problem is represented as the emptiness problem of a Büchi automaton which turns out to be equivalent to finding a reachable accepting cycle in the underlying graph. The Büchi automaton is obtained as a synchronous product of two automata: the one modelling the given system and the other one representing the negation of the verified property. The *nested depth-first search* algorithm is used to discover an accepting cycle in the graph. The pseudo-code of the algorithm is given in Figure 1.

We use the decomposition of the formula automaton into strongly connected components as a heuristic to partition the state space. The main idea is that the partition function checks to which maximal strongly connected component the formula member of a state belongs to and the state is placed on the same network node as the other states of the component. The partition function is static and can effectively be pre-computed.

The rest of the paper is organized as follows. We start with a section which briefly reviews the basics of automata-based LTL model checking. In the following section we describe our approach to the partitioning of the state space and present the distributed nested depth-first search algorithm. In the next section we report some experimental results.

## 2 Distributed Nested DFS Algorithm

Our aim is to distribute the standard (sequential) nested DFS algorithm on a network of workstations that communicate via message-passing. The algorithm was chosen not only because of its efficiency, but also because a distribution of this algorithm seems to be a natural extension of commonly used verification tools such as SPIN. The other reason was that we would like to improve our two distributed algorithms for LTL model checking ([2, 5]). Both of them have used partition functions that randomly distribute the state space among the network nodes. The new technique presented in this paper is independent of them, hence, it provides their natural supplementary improvement.

A "naive" approach to the distribution is obvious. The graph can be partitioned into several parts among the network nodes, each network node owns a subset of the graph (state-space), and the algorithm uses some kind of baton to transfer the computation among the network nodes. Only one network node is active at a time and thus no parallelism occurs. If a state belonging to another network node is reached then the algorithm passes the baton to the owner of the state, i.e., the computation is interrupted and the execution is transfered over the network to another network node which continues the search and returns the execution back after exploring the whole appropriate subgraph. The edge in the graph which causes such transfer over network is called *cross edge* or cross transition. This method merely extends the amount of available random access memory, but no speed up can be achieved.

The other extreme approach to the distribution of the nested DFS algorithm is to allow simultaneous (parallel) execution of the nested DFS algorithm on each network node with a randomly partitioned state-space. However, such an approach to the model checking could lead to an incorrect result since the depth-first *order* of visits of states is not preserved. The order in which the states are visited does not matter in the case of safety properties (the problem can be reduced to the reachability problem).

Our aim is to partition the graph in such a way that no (reachable) accepting cycle (passing through an accepting state) is splitted among more network nodes, i.e., all states in the cycle belong to the same network node. Therefore, there is no state that could be visited by two different nested searches originating from different network nodes. The question is whether it is possible to find such a partition of the graph effectively and whether the partition fulfils the two basic aspects of distributed algorithm which are the locality and balance.

A safe solution is to decompose the graph into strongly connected components first and then to partition the graph according to this decomposition. In addition, the decomposition can make the nested (second) search more efficient by searching only those paths that can really form a cycle in the graph (i.e., the paths that belong to one maximal strongly connected component). However, decomposing the system in advance would actually solve the verification problem.

In the case of automata-based LTL model checking the underlying graph (which is supposed to be searched for cycles) is a graph of a *product* automaton. This product automaton is a result of a synchronisation of the small *negative claim automaton* with a huge *system automaton* (with all states considered as accepting). The system automaton models the behaviour of the given system and the negative claim automaton describes the behaviour which contradicts the verified property. We can utilise the information gained from the negative claim automaton to decompose the product graph into *sets* of maximal strongly connected components. Though this is not the ideal solution, it still results in an effective nested search and can lead to a reasonable distribution of the problem.

In contrast to the product automaton it is possible to find effectively all maximal strongly connected components of the negative claim automaton and to use them to decompose the product automaton into sets of maximal strongly connected components. As the nested search "remains" on one network node the level of asynchronous behaviour of the algorithm is increased.

**Definition 1.** *Let* $A = (\Sigma, S_A, q_A, \delta_A, F_A)$ *and* $B = (\Sigma, S_B, q_B, \delta_B, F_B)$ *be Büchi automata. The* synchronous product $A \otimes B$ *of* $A$ *and* $B$ *is the automaton* $(\Sigma, S, q, \delta, F)$, *where* $S = S_A \times S_B, q = (q_A, q_B), F = F_A \times F_B$ *and* $(u', v') \in \delta((u, v), a)$ *if and only if* $u' \in \delta_A(u, a)$ *and* $v' \in \delta_B(v, a)$. *The automata* $A$ *and* $B$ *are called* projections.

We identify a Büchi automaton with its graph representation. Let $G$ be a graph of a Büchi automaton and let $s$ be a state (vertex) of this graph. Then we denote by $SCC_G(s)$ the maximal strongly connected component of the graph $G$ that the state $s$ belongs to and by $SCC(G)$ the set of all maximal strongly connected components of the graph $G$.

For $s = (s_A, s_B) \in A \otimes B$ we put $\pi_1(s) = s_A$ and $\pi_2(s) = s_B$ and for $C \subseteq A \otimes B$ we define $\pi_1(C) = \bigcup_{s \in C} \pi_1(s)$ and $\pi_2(C) = \bigcup_{s \in C} \pi_2(s)$ ($\pi_1, \pi_2$ are called *projections*). Finally, we can define the function $Part(s) : S_{A \otimes B} \to 2^{S_A}$ as

$$Part(s) = SCC_G(\pi_1(s))$$

In the following we suppose that the graph $G$ is a synchronous product of automata $A$ and $B$. We now state some properties of strongly connected components our distributed version of the nested DFS algorithm is based on.

**Lemma 1.** *For each maximal strongly connected component in* $B$ *there is a corresponding set of maximal strongly connected components in* $A \otimes B$.

The following lemma says that a strongly connected component in a product automaton corresponds to strongly connected components in its two projections.

**Lemma 2.** *Let* $A, B$ *be Büchi automata. If* $\mathcal{C} \in SCC(A \otimes B)$ *then* $\pi_1(C)$ *forms a (not necessarily maximal) strongly connected component of* $A$ *and* $\pi_2(C)$ *forms a (not necessarily maximal) strongly connected component of* $B$.

It follows immediately that for any cycle $\mathcal{C} = s_0, \ldots, s_n, s_0$ in $G$ the equality $Part(s_0) = Part(s_1) = \ldots = Part(s_n) = Part(s_0)$ holds. Moreover, we claim the following lemma.

**Lemma 3.** *Let* $A, B$ *be Büchi automata. If* $\mathcal{D}$ *in* $A \otimes B$ *is an accepting cycle then the set* $\pi_1(\mathcal{D})$ *forms an accepting cycle in* $A$ *and the set* $\pi_2(\mathcal{D})$ *forms an accepting cycle in* $B$.

According to Lemma 3 it is meaningful to characterise the types of maximal strongly connected components of a given graph. Maximal strongly connected components can be classified in the following three categories:

**Type F:** (*Fully Accepting*) Any cycle within the component contains at least one accepting state. (There is no non-accepting cycle within the component.)
**Type P:** (*Partially Accepting*) There is at least one accepting cycle and one non-accepting cycle within the component.
**Type N:** (*Non-Accepting*) There is no accepting cycle within the component.

We define three predicates on states of the graph $G$. $F\text{-}SCC_G(s)$, $P\text{-}SCC_G(s)$ and $N\text{-}SCC_G(s)$ is true if and only if $SCC_G(s)$ is *Fully accepting*, *Partially accepting* or *Non-accepting* component respectively.

**Lemma 4.**

1. *Let $s \in A \otimes B$ such that $F\text{-}SCC_G(\pi_1(s))$ and $F\text{-}SCC_G(\pi_2(s))$. Then any cycle in $A \otimes B$ containing the state $s$ is an accepting cycle.*
2. *Let $s \in A \otimes B$ such that $N\text{-}SCC_G(\pi_1(s))$ or $N\text{-}SCC_G(\pi_2(s))$. Then any cycle in $A \otimes B$ containing the state $s$ is non-accepting.*

Note that in the remaining cases we cannot say anything about the "type" of the cycle.

We now present the distributed version of the nested DFS algorithm which effectively uses the decomposition of the negative claim automaton $A$ into maximal strongly connected components and exploits the type of each component. Note, that in the Büchi automaton that models the system all the state are accepting.

The strategy for partitioning the state space of the product automaton places all states that belong to the same component on the same network node regardless of the component's type. The advantage is that a node can be considered to be of an appropriate "type", hence different and specific algorithms can be used on different nodes.

We further modify the basic algorithm from Figure 1 in the following way. The situation, when the currently explored state $s$ has already been put in *in_stack* and the predicate $F\text{-}SCC_G(\pi_1(s))$ is true, ensures the presence of an accepting cycle (see Lemma 4.1). Therefore, the nested search need not be called in this case at all.

Moreover, we can restrict the nested DFS only to those successors which projection on the never claim automaton is in the same strongly connected component, making the nested DFS even more efficient.

The pseudo-code of the final Distributed Nested DFS algorithm (which is running on each network node) is given in Figure 2. Each node maintains its own local queue of states to be explored and is responsible for its own part of the state space. When a node computes a new state $s$, it uses the function `Part(s)` to find out whether the state belongs to the state subset of the network node. If the state is local the node continues locally, otherwise a message containing the state is sent to the owner node of the state.

## 3 Implementation and Experiments

We have implemented an experimental version of the Distributed Nested DFS algorithm using the SPIN verifier version 3.4.10 and performed a series of preliminary tests. The implementation has been done in C using our own version of a simple message-passing layer over the standard TCP/IP. The distributed model checking requires effective *message buffering* to achieve better practical performance and e.g. the MPI library is due its generality not suitable as it consumes too much computational recourses. All the experiments were performed without partial order reductions on a cluster of Pentium PC Linux workstations with 256 Mbytes of RAM each, interconnected with a 100Mbps Ethernet.

```
proc Node(i)
  if Part(initstate) = i
    then queue[i] := {initstate}
     else queue[i] := ∅
  fi
  while Not End do
        if queue[i] ≠ ∅
          then state := Head(queue[i])
               queue[i] := Tail(queue[i])
               in_stack[i] := {state}
               DFS(i, state)
        fi
  od
end


proc DFS(i, state)
  if (state, 0) ∉ visited[i]
    then visited[i] := visited[i] ∪ {(state, 0)};
         foreach s ∈ Succ(state) do
           if Part(s) ≠ i
             then queue[Part(s)] := (queue[Part(s)], s)
              else in_stack[i] := in_stack[i] ∪ {s}
                   DFS(i, s)
                   in_stack[i] := in_stack[i] \ {s}
           fi od
         if Accepting(state) ∧ Part(state) is of type P
           then NestedDFS(i, state)
         fi
     else if state ∈ in_stack[i] ∧ Part(state) is of type F
           then Report("Cycle Found")
         fi fi
end


proc NestedDFS(i, state)
  if (state, 1) ∉ visited
    then visited := visited ∪ {(state, 1)}
         foreach s ∈ Succ(state) do
           if Part(s) = Part(state)
             then if state ∈ in_stack[i]
                    then Report("Cycle Found")
                     else NestedDFS(i, s)
                  fi fi
         od fi
end
```

**Fig. 2.** Distributed Nested DFS Algorithm

Our algorithm uses a manager process running on the "master" network node to initiate and terminate the distributed computation. The manager process decomposes the negative claim automaton into maximal strongly connected components using the standard Tarjan's algorithm [12] and defines the partition of the state space. In the current implementation we assign the part corresponding to one maximal strongly connected component to one network node regardless of the component type.

In the distributed computation all the involved network nodes perform the same algorithm for the assigned part of the state space. The termination detection is done using a virtual ring-based distributed algorithm.

For our experiments we have decided to choose some of the standard examples found in the SPIN distribution (*leaders, ABP*) and in the HSF-SPIN distribution (*elevator*) together with some others (*elevator II*). The leader election protocol and the elevator model are parametrised and the size of the state space is exponential in the parameters.

For each example specified as a PROMELA model we have performed four experiments. We have run the sequential nested DFS algorithm (SPIN) to verify the property (results are reported in the column "NDFS") and standard DFS algorithm (SPIN) to generate the reachable state space (column "DFS"). The latter is important even if the the property does not hold in order to obtain the necessary information about the size of the whole reachable state space. The remaining columns report results of distributed nested DFS computation (column "D-NDFS") and "naively" (only one node active at a time) distributed standard DFS computation (column "D-DFS"). Each sub-column of columns "D-NDFS" and "D-DFS" records the results obtained for one network node.

We did not compare the results to any other distributed methods or techniques for the LTL model checking since the newly presented approach is independent of them. The results have to be interpreted as a possible supplementary way of the state-space partition that can be combined with most of the previously presented techniques and methods.

Every table contains for each network node the following information: the number of *visited states*, the number of *transitions*, and the number of sent *messages*. As messages are buffered the *number of packets* is also reported. *Accepting cycle* indicates whether an accepting cycle was discovered on the network node or not. *Runtime* gives the running time of the experiment in seconds.

### 3.1 Alternating Bit Protocol

ABP is a very simple and small system. The purpose of this experiment was to evaluate how the network communication and book-keeping slow-downs the overall performance. Moreover, we were quite surprised by the perfectly balanced partition of the state space among the network nodes.

| | |
|---|---|
| Model: | Alternating Bit Protocol |
| Parameters: | none |
| Formula: | $\Box(p \Rightarrow ((\Diamond q) \vee (\Diamond r)))$ |
| The number of maximal SCC: | 3 |

| | NDFS | DFS | D-NDFS | | | D-DFS | | |
|---|---|---|---|---|---|---|---|---|
| Visited states | 24 | 69 | 23 | 10 | 9 | 23 | 23 | 23 |
| Transitions | 32 | 110 | 34 | 10 | 9 | 34 | 38 | 38 |
| Messages | — | — | 12 | 3 | 3 | 12 | 2 | 2 |
| Number of packets | — | — | 4 | 3 | 3 | 4 | 2 | 2 |
| Accepting cycle | yes | — | no | yes | yes | — | — | — |
| Runtime (seconds) | 0.02 | 0.01 | 1.33 | | | 1.07 | | |

### 3.2 Elevator

The elevator has no strategy for serving the floors as the next floor to be served is chosen randomly. That is why hardly any formula representing the standard elevator's property holds. The model has been tested for several values of the parameter LEVELS. For bigger values of the parameter the distributed algorithm was even faster than the original sequential one in spite of the communication overhead, however, the speedup was not the primary goal of the approach.

Another remarkable fact is that the proportion of number of visited states on the network nodes is preserved in models with different parameter. We suppose this is caused by the natural "similarity" or "regularity" of the reachable state spaces.

| | |
|---|---|
| Model: | Elevator |
| Parameters: | LEVELS = 3 (the number of floors) |
| Formula: | $\Box(p \Rightarrow \Diamond(q \wedge r))$ |
| The number of maximal SCC: | 2 |

| | NDFS | DFS | D-NDFS | | D-DFS | |
|---|---|---|---|---|---|---|
| Visited states | 223 | 218327 | 35649 | 167 | 142597 | 75730 |
| Transitions | 397 | 941279 | 141108 | 210 | 564434 | 376845 |
| Messages | — | — | 27817 | 2 | 111271 | 2 |
| Number of packets | — | — | 278 | 2 | 1115 | 2 |
| Accepting cycle | yes | — | no | yes | — | — |
| Runtime (seconds) | 0.06 | 3.17 | 1.18 | | 4.09 | |

| | |
|---|---|
| Model: | Elevator |
| Parameters: | LEVELS = 4 (the number of floors) |
| Formula: | $\Box(p \Rightarrow \Diamond(q \wedge r))$ |
| The number of maximal SCC: | 2 |

| | NDFS | DFS | D-NDFS | | D-DFS | |
|---|---|---|---|---|---|---|
| Visited states | 246 | 1542810 | 48162 | 193 | 963240 | 579569 |
| Transitions | 420 | 7894380 | 223118 | 366 | 4462360 | 3432020 |
| Messages | — | — | 47791 | 2 | 955824 | 2 |
| Number of packets | — | — | 48 | 2 | 958 | 2 |
| Accepting cycle | yes | — | no | yes | — | — |
| Runtime (seconds) | 0.07 | **45.35** | 2.13 | | **39.73** | |

### 3.3 Elevator II

In contrast to the previous example the Elevator II serves the floors in a fair way. This allows to formulate interesting liveness properties, e.g. that the elevator passes every floor without serving it at most once. The decomposition of the negative claim automaton has 11 maximal strongly connected components. In the table we present the network nodes with the maximal and the minimal number of visited states only. Please note, the same formula can be verified as two standalone formulas (emergent by splitting the formula in the top-level wedge) each represented by an automaton with 8 maximal strongly connected components.

| | |
|---|---|
| Model: | Elevator II |
| Parameters: | LEVELS = 5 (the number of floors) |
| Formula: | $\Box(r_0 \rightarrow (\neg p_0\ U\ (p_0\ U\ (\neg p_0\ U\ (p_0 \wedge o)))))\wedge$ $\Box(r_1 \rightarrow (\neg p_1\ U\ (p_1\ U\ (\neg p_1\ U\ (p_1 \wedge o)))))$ |
| The number of maximal SCC: | 11 |

| | **NDFS** | **DFS** | **D-NDFS** | | **D-DFS** | |
|---|---|---|---|---|---|---|
| | | | max | min | max | min |
| Visited states | 542324 | 542324 | 119773 | 0 | 119773 | 0 |
| Transitions | 5474040 | 2566650 | 560074 | 0 | 560074 | 0 |
| Messages | — | — | 405429 | 3 | 405428 | 2 |
| Number of packets | — | — | 1357 | 3 | 1356 | 2 |
| Accepting cycle | no | — | no | no | — | — |
| Runtime (seconds) | **28.35** | 15.89 | **19.29** | | 19.59 | |

## 4 Conclusions

We have proposed a distributed algorithm for LTL model checking that runs on a cluster of PCs. The main novelty of our approach is that we use the decomposition of the negative claim automaton into maximal strongly connected components to distribute the verification problem over the cluster. In addition to the fact that we are able to decompose the task so that several instances of the verification procedure can be performed in parallel, we are also able to perform an improved version of the nested DFS algorithm. We stress that our technique is compatible with other state space saving techniques, like partial order reductions, state compression, state hashing. And it can perfectly supplement already existing methods and techniques for state-space partition in spite of the fact that the technique is not quite proper for standalone usage as the number of maximal strongly connected components of the claim automaton is usually small. The new approach to the distribution of the algorithm can be used in the framework of multi-thread programming as well.

There are other possible strategies for partitioning the state space which employ to some extent knowledge about the type of components. One of them places states belonging to a component of type $N$ randomly on network nodes. This is possible because the only relevant information for these states is their reachability and can be analysed e.g. using the algorithm of Lerda and Sisto[10]. The other parts of graph which are of type $P$ and $F$ can be distributed using algorithm presented in [2] or [5] or using the simplest baton algorithm.

In the future we intend to implement and experiment other strategies for distribution of the verification problem that uses additional information from the verified property. Also, we would like to continue our search for similar improvements achieved by the structure gained from the modelled system.

The other question is whether it is possible to find specialised algorithms for Fully accepting and Partially accepting subclasses of the problem. We would like also to explore the possibility and cost of turning all type $P$ components of negative claim automaton into type $F$ components which can lead to simpler algorithm.

## References


1. S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In G. Bilardi, A. G. Ferreira, R. Lüling, and J. D. P. Rolim, editors, *Proceeding of the Parallel Computing Conference PARCO'97 (Bonn, Germany)*, volume 1253 of *LNCS*, pages 207–218. Springer, 1997.
2. J. Barnat, L. Brim, and J. Stříbrná. Distributed ltl model-checking in SPIN. In Matthew B. Dwyer, editor, *8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.
3. B. Bollig, M. Leucker, and M Weber. Parallel model checking for the alternation free mu-calculus. In T. Margaria and W. Yi, editors, *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.
4. L. Brim, D. Gilbert, J-M. Jacquet, and M. Křetínský. Multi-agent systems as concurrent constraint processes. In L. Pacholski and P. Ružička, editors, *SOFSEM'01*, number 2234 in Lecture Notes in Computer Science, pages 201–210. Springer Verlag, 2001.
5. L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL Model Checking Based on Negative Cycle Detection. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FST TCS 2001*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.
6. G. Ciardo, J. Gluckman, and D.M. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 1997.
7. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.
8. F. de Boer, R. van Eijk, M. van der Hoek, and Ch. Meyer. Failure semantics for the exchange of information in multi-agent systems. In *CONCUR: 11th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2000.
9. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In Matthew B. Dwyer, editor, *8th International SPIN Workshop*, number 2057 in LNCS, pages 57–79. Springer, 2001.
10. F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proc. 6th SPIN Workshop on Model Checking of Software (SPIN99)*, volume 1680 of *LNCS*. Springer, 1999.
11. D.M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-space Generation. Technical Report NASA/CR-2000-210082, NASA Langley Research Center, Hampton, USA, 2000.
12. Robert Tarjan. Depth first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.
13. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, Washington, DC, 1986.