Parallel State Space Construction for Model-Checking

Hubert Garavel, Radu Mateescu, and Irina Smarandache

INRIA Rhône-Alpes / VASY, 655, avenue de l'Europe
F-38330 Montbonnot Saint Martin, France
Hubert.Garavel@inria.fr, Radu.Mateescu@inria.fr, Irina.Sturm@st.com

Abstract. The verification of concurrent finite-state systems by model-checking often requires to generate (a large part of) the state space of the system under analysis. Because of the state explosion problem, this may be a resource-consuming operation, both in terms of memory and CPU time. In this paper, we aim at improving the performances of state space construction by using parallelization techniques. We present parallel algorithms for constructing state spaces (or Labeled Transition Systems) on a network or a cluster of workstations. Each node in the network builds a part of the state space, all parts being merged to form the whole state space upon termination of the parallel computation. These algorithms have been implemented within the CADP verification tool set and experimented on various concurrent applications specified in LOTOS. The results obtained show close to ideal speedups and a good load balancing between network nodes.

Key-words: distributed algorithms, labeled transition system, LOTOS, model-checking, state space construction, verification

1 Introduction

As formal verification becomes increasingly used in the industry as a part of the design process, there is a constant need for efficient tool support to deal with real-size applications. Model-checking [20, 10] is a successful verification method based on reachability analysis (state space exploration) and allows an automatic detection of early design errors in finite-state systems. Model-checking works by constructing a model (state space) of the system under design, on which the desired correctness properties are verified.

There are essentially two approaches to model-checking: *symbolic* verification [9, 10] represents the state space in comprehension, by using various encoding techniques (e.g., BDDs), and *enumerative* verification [32, 11, 12, 19] represents the state space in extension, by enumerating all reachable states. Enumerative model-checking techniques can be further divided in *global* techniques, which require to entirely construct the state space before performing the verification, and *local* (or *on-the-fly*) techniques, which allow to construct the state space simultaneously with the verification.

In this paper, we focus on enumerative model-checking, which is well-adapted to asynchronous, non-deterministic systems containing complex data types (records, sets, lists, trees, etc.). More precisely, we consider the problem of constructing a *Labeled Transition System* (LTS), which is the natural model for high-level, action-based specification languages, especially process algebras such as CCS [30], CSP [18], ACP [4], or LOTOS [21]. An LTS is constructed by exploring the transition relation starting from the initial state (*forward reachability*). During this operation, all explored states must be kept in memory in order to avoid multiple exploration of a same state. Once the LTS is constructed, it can be used as input for various verification procedures, such as bisimulation/preorder checking and temporal logic model-checking. Moreover, when the verification requires to explore the entire LTS (e.g., when verifying invariant temporal properties or checking bisimulation), since the state contents is abstracted away in a constructed LTS, the memory consumed is generally much smaller than for on-the-fly verification on the initial specification.

State space construction may be very consuming both in terms of memory and execution time: this is the so-called *state explosion* problem. During the last decade, different techniques for handling state explosion have been proposed, among which partial orders and symmetries; however, for industrial-scale systems, these optimizations are not always sufficient. Moreover, most of the currently available verification tools work on sequential machines, which limits the amount of memory (between 0.5 and 2 GBytes on usual configurations), and therefore the use of clusters or networks of workstations is desirable.

In this paper, we investigate an approach to parallelize state space construction on several machines, in order to benefit from all the local memories and CPU resources of each machine. This allows to reduce both the amount of memory needed on each machine and the overall execution time. We propose algorithms for parallel construction of LTSs, developed using the generic environments BCG and OPEN/CÆSAR [13] for LTS manipulation provided by the CADP verification tool set [12]. Since these environments are language independent, our algorithms can be directly used not only for LOTOS, but also for every language connected to the OPEN/CÆSAR application programming interface, such as UML [22].

The implementation is based on standard sockets, available everywhere, and was experimented on two different configurations: a typical network of workstations (Sparc workstations running Solaris and PCs running Linux, connected using 100 Mb/s Ethernet), and a cluster of PCs (with 450 MHz processor and 512 MBytes main memory) connected using SCI (*Scalable Coherent Interface*). Each machine in the network is responsible for constructing a part of the LTS, this part being determined using a static partition function. Upon termination of the parallel computation, which is detected by means of a virtual ring-based distributed algorithm, all parts are merged to form the complete LTS.

We experimented with our algorithms on three non-trivial protocols specified in LOTOS: the home audio-video (HAVi) protocol of Philips [33], the TOKENRING leader election protocol [14], and the SCSI-2 bus arbitration protocol [3].

Related work Distributed state space construction has been studied in various contexts, mostly for the analysis of low-level formalisms. such as Petri nets, stochastic Petri nets, discrete-time and continuous-time Markov chains [5, 6, 2, 1, 8, 31, 27, 16, 23].

All these approaches share a common idea: each machine in the network explores a subset of the state space. However, they differ on a number of design principles and implementation choices such as: the choice between a shared memory architecture and a message-passing one, the use of hash tables or B-trees to store states on each machine, the way of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, etc.

As regards high-level languages for asynchronous concurrency, a distributed state space exploration algorithm [26] derived from the SPIN model-checker [19] has been implemented for the PROMELA language. The algorithm performs well on homogeneous networks of machines, but it does not outperform the standard, sequential implementation of SPIN, except for problems that do not fit into the main memory of a single machine. Several SPIN-specific partition functions are experimented, the most advantageous one being a function that takes into account only a fraction of the state vector.

Another distributed state enumeration algorithm has been implemented in the MURφ verifier [34]. The speedups obtained are close to linear and the hash function used for state space partition provides a good load balancing. However, experimental data reported concerns relatively small state spaces (approximatively 1.5 M states) on a 32-node UltraSPARC Myrinet network of workstations.

There also exist approaches, such as [24], in which parallelization is applied to "partial" verification, i.e., state enumeration in which some states can be omitted with a low probability. In the present paper, we only address exact, exhaustive verification issues.

For completeness, we can also mention an alternative approach [17] in which symbolic reachability analysis is distributed over a network of workstations: this approach does not handle states individually, but sets of states encoded using BDDs.

Paper outline Section 2 gives some preliminary definitions and specifies the context of our work. Section 3 describes the proposed algorithms for parallel construction of LTSs. Section 4 discusses implementation issues and presents various experimental results. Finally, Section 5 gives some concluding remarks and directions for future work.

2 Definitions

A (monolithic) Labeled Transition System (LTS) is a tuple $M = (S, A, T, s_0)$, where S is the set of states, A is the set of actions, $T \subseteq S \times A \times S$ is the transition relation, and $s_0 \in S$ is the initial state. A transition $(s, a, s') \in T$ indicates that the system can move from state s to state s' by performing action a. All states in S are assumed to be reachable from s_0 via (sequences of) transitions in T.

In the model-checking approach by state enumeration, there are essentially two ways to represent an LTS:

explicitly, by enumerating all its states and transitions. In this case, the contents of states becomes irrelevant, since the essential information is given by actions (transition labels). Therefore, when storing an LTS as a computer file, it is sufficient to encode states as natural numbers. An explicit representation of LTSs is provided by the BCG (*Binary Coded Graph*) file format of the CADP verification tool set [12]. The BCG format is based upon specialized compression algorithms, allowing compact encodings of LTSs.

implicitly, by giving its initial state s_0 and its successor function $succ : S \to 2^T$ defined by $succ(s) = \{(s, a, s') \mid (s, a, s') \in T\}$. An implicit representation of LTSs is provided by the generic, language independent environment OPEN/CÆSAR [13] of CADP. OPEN/CÆSAR offers primitives for accessing the initial state of an LTS and for enumerating the successors of a given state, as well as various data structures (state tables, stacks, etc.), allowing straightforward implementations of on-the-fly verification algorithms.

Our objective is to translate LTSs from an implicit to an explicit representation by using parallelization techniques.

In order to represent a monolithic LTS $M = (S, A, T, s_0)$ on N machines (numbered from 0 to $N-1$), we introduce the notion of *partitioned* LTS $D = (M_0, \ldots, M_{N-1}, s_0)$, where: $S = \cup_{i=0}^{N-1} S_i$ and $S_i \cap S_j = \emptyset$ for all $0 \leq i, j < N$ (the state set is partitioned into N classes, one class per machine), $A = \cup_{i=0}^{N-1} A_i$, $T = \cup_{i=0}^{N-1} T_i$ and $(s, a, s') \in T_i \Rightarrow s' \in S_i$ for all $0 \leq i < N$ (transitions between two states belonging to different classes are part of the transition relation of the component LTS containing the target state). Note that initial states of the component LTSs M_i are irrelevant, since the corresponding subgraphs may be not connected (i.e., not reachable from a single state). A partitioned LTS can be represented as a collection of BCG files encoding the components $M_0, \ldots, M_{N-1}$.

3 Parallel generation of LTSs

In this section we present two complementary algorithms allowing to convert an implicit LTS (defined using the OPEN/CÆSAR interface) to an explicit one (represented as a BCG file) using N machines connected by a network. These algorithms operate in two steps:

– Construction of a partitioned LTS represented as a collection of BCG files. This is done by using an algorithm called DISTRIBUTOR, which is executed on every machine in order to generate a BCG file encoding a component of the partitioned LTS.
– Conversion to a monolithic LTS represented as a single BCG file. This is done using an algorithm called BCGMERGE, which is executed on a sequential machine in order to generate a single BCG file containing all the states and transitions of the partitioned LTS.

Once the BCG file encoding the initial LTS has been constructed, it can be used as input for the EVALUATOR 3.0 model-checker [28] of CADP, which allows linear-time verification of temporal formulas expressed in regular alternation-free μ-calculus.

3.1 Construction of partitioned LTSs

We consider a network of N machines numbered from 0 to $N-1$ and an LTS $M = (S, A, T, s_0)$ given implicitly by its initial state s_0 and its successor function *succ*. Machine i can send a message m to machine j by invoking a primitive named SEND (j, m), and can receive a message by invoking another primitive RECEIVE (m). There are four kinds of messages: *Arc*, *Rec*, *Snd*, and *Trm*, the first one being used for sending LTS transitions and the others being related to termination detection. SEND and RECEIVE are assumed to be non-blocking. RECEIVE returns a boolean answer indicating whether a message has been received or not.

The parallel generation algorithm DISTRIBUTOR that we propose is shown on Figure 1. Each machine executes an instance of DISTRIBUTOR and explores a part of the state space S. The states explored by each machine are determined using a static partition function $h : S \rightarrow [0, N-1]$. Machine i explores all states s such that $h(s) = i$ and produces a BCG file $B_i = (S_i, A_i, T_i)$. The computation is started by the machine called *initiator*, having the index $h(s_0)$, which explores the initial state of the LTS.

The states visited and explored by machine i during the forward traversal of the LTS are stored in two sets V_i ("visited") and E_i ("explored"), respectively. V_i and E_i are implemented using the state table handling primitives provided by the OPEN/CÆSAR environment. The transitions to be written to the local BCG file B_i are temporarily kept in a work list L_i. It is worth noticing that the DISTRIBUTOR algorithm only keeps in memory the state set S_i of the corresponding component of the partitioned LTS, whilst the transition relation T_i is stored in the BCG file B_i. The DISTRIBUTOR algorithm consists of a main loop, which performs three actions:

(a) A state $s \in V_i$ is explored by enumerating all its successor transitions $(s, a, s') \in succ(s)$. If a target state s' belongs to machine i (i.e., $h(s') = i$), the corresponding transition is kept in the list L_i and will be processed later. Otherwise, the transition is sent to machine $h(s')$ as a message $Arc(n_i(s), a, s')$, where $n_i(s)$ is the number associated by machine i to s. Machine $h(s')$ is responsible for writing the transition to its local BCG file and for exploring state s'. Note that there is no need to send the contents of state s itself, but only its number $n_i(s)$.

(b) A transition is taken from L_i and is written to the BCG file B_i by computing appropriate numbers for the source and target states. In order to obtain a bijective numbering of LTS states across the N BCG files, each state s explored by machine i is assigned a number $n_i(s)$ such that $n_i(s)\ mod\ N = i$. This is done using a counter c_i, which is initialized to i and incremented by N every time a new state is visited.

```
procedure DISTRIBUTOR ($i$, $s_0$, $succ$, $h$, $N$) is
    $initiator_i := (h(s_0) = i)$; $L_i := \emptyset$; $E_i := \emptyset$; $A_i := \emptyset$; $T_i := \emptyset$; $c_i := i$;
    if $initiator_i$ then
        $terminit := false$; $n_i(s_0) := c_i$; $V_i := \{s_0\}$; $S_i := \{n_i(s_0)\}$
    else
        $V_i := \emptyset$; $S_i := \emptyset$
    endif;
    $terminated_i := false$; $nbsent_i := 0$; $nbrecd_i := 0$;
    while $\neg terminated_i$ do
(a)     if $V_i \neq \emptyset$ then
            let $s \in V_i$; $V_i := V_i \setminus \{s\}$; $E_i := E_i \cup \{s\}$;
            forall $(s, a, s') \in succ(s)$ do
                if $h(s') = i$ then
                    $L_i := L_i \cup \{(n_i(s), a, s')\}$
                else
                    SEND ($h(s')$, $Arc(n_i(s), a, s')$); $nbsent_i := nbsent_i + 1$
                endif
            endfor
(b)     elsif $L_i \neq \emptyset$ then
            let $(n, a, s) \in L_i$; $L_i := L_i \setminus \{(n, a, s)\}$;
            if $s \notin E_i \cup V_i$ then
                $c_i := c_i + N$; $n_i(s) := c_i$; $V_i := V_i \cup \{s\}$; $S_i := S_i \cup \{n_i(s)\}$;
            endif;
            $A_i := A_i \cup \{a\}$; $T_i := T_i \cup \{(n, a, n_i(s))\}$
        endif;
(c)     if RECEIVE ($m$) then
            case $m$ is
                $Arc(n, a, s) \rightarrow L_i := L_i \cup \{(n, a, s)\}$; $nbrecd_i := nbrecd_i + 1$
                $Rec(k) \rightarrow$ if $\neg initiator_i$ then
                                SEND ($(i + 1)$ $mod$ $N$, $Rec(k + nbrecd_i)$)
                            elsif $terminit$ then
                                $totalrecd := k$; SEND ($(i + 1)$ $mod$ $N$, $Snd(nbsent_i)$)
                            endif
                $Snd(k) \rightarrow$ if $\neg initiator_i$ then
                                SEND ($(i + 1)$ $mod$ $N$, $Snd(k + nbsent_i)$)
                            elsif $terminit \wedge totalrecd = k$ then
                                SEND ($(i + 1)$ $mod$ $N$, $Trm$)
                            else
                                $terminit := false$
                            endif
                $Trm \rightarrow$     if $\neg initiator_i$ then
                                SEND ($(i + 1)$ $mod$ $N$, $Trm$)
                            endif;
                            $terminated_i := true$
            endcase
        endif;
        if $L_i = \emptyset \wedge V_i = \emptyset \wedge initiator_i \wedge \neg terminit$ then
            $terminit := true$; SEND ($(i + 1)$ $mod$ $N$, $Rec(nbrecd_i)$)
        endif
    endwhile
end
```

Fig. 1. Parallel generation of an LTS as a collection of BCG files

(c) An attempt is made to receive a message m from another machine. If m has the form $Arc(n, a, s)$, it denotes a transition (s', a, s), where n is the source state number $n_j(s')$ assigned by the sender machine of index $j = n \ mod \ N$. In this case, the contents of m is stored in the list L_i; otherwise, m is related to termination detection (see below). Thus, the BCG file B_i will contain all LTS transitions whose target states are explored by machine i.

In order to detect the termination of the parallel LTS generation, we use a virtual ring-based algorithm inspired by [29]. According to the general definition, (global) termination is reached when all local computations are finished (i.e., each machine i has neither remaining states to explore, nor transitions to write in its BCG file B_i) and all communication channels are empty (i.e., all sent transitions have been received).

The principle of the termination detection algorithm used in DISTRIBUTOR is the following. All machines are supposed to be on an unidirectional virtual ring that connects every machine i to its successor machine $(i + 1) \ mod \ N$. Every time the initiator machine finishes its local computations, it checks whether global termination has been reached by generating two successive waves of *Rec* and *Snd* messages on the virtual ring to collect the number of messages received and sent by all machines, respectively. A message $Rec(k)$ (resp. $Snd(k)$) received by machine i indicates that k messages have been received (resp. sent) by the machines from the initiator up to $(i - 1) \ mod \ N$. Each machine i counts the messages it has received and sent using two integer variables $nbrecd_i$ and $nbsent_i$, and adds their values to the numbers carried by *Rec* and *Snd* messages. Upon receipt of the $Snd(k)$ message ending the second wave, the initiator machine checks whether the total number k of messages sent is equal to the total number *totalrecd* of messages received (the result of the *Rec* wave). If this is the case, it will inform the other machines that termination has been reached, by sending a *Trm* message on the ring. Otherwise, the initiator concludes that termination has not been reached yet and will generate a new termination detection wave later.

In practice, to reduce the number of termination detection messages, each machine propagates the current wave only when its local computations are finished (for simplicity, we did not specify this in Figure 1). Experimental results have shown that in this case there is almost no termination detection overhead, two waves being always sufficient. This distributed termination detection scheme seems to use less messages than the centralized termination detection schemes used in the parallel versions of SPIN [26] and MURφ [34], which in all cases require several broadcast message exchanges between a coordinator machine and all other machines.

3.2 Merging of partitioned LTSs into monolithic LTSs

After constructing a collection of N BCG files representing a partitioned LTS by using the DISTRIBUTOR algorithm, the next step is to convert them into a unique BCG file in order to make it usable by the verification tools of CADP.

Since the states contained in different BCG files have been given unique numbers by the DISTRIBUTOR algorithm (i.e., every state belonging to the BCG file B_i has an index k such that $k \ mod \ N = i$ and two states belonging to the same BCG file have different numbers), this could simply be done by concatenating all transitions of the N BCG files.

However, since the partition function h is not perfect, a simple concatenation may result in a BCG file with an initial state number different from 0 (when $h(s_0) \neq 0$) and with "holes" in the numbering of states (when $|S_i| \neq |S_j|$ for two BCG files B_i and B_j). For example, for an LTS with 7 states and $N = 2$, DISTRIBUTOR could produce $S_0 = \{0, 2, 4, 6, 8\}$, $S_1 = \{1, 3\}$, and $h(s_0) = 1$, which would lead by concatenation to a BCG file with $S = \{0, 1, 2, 3, 4, 6, 8\}$ instead of $S = \{0, 1, 2, 3, 4, 5, 6\}$. A contiguous renumbering of the states would be more suitable for achieving a better compaction of the final BCG file.

The conversion algorithm BCGMERGE that we propose (see Figure 2) takes as inputs a partitioned LTS represented as a collection of BCG files $B_0, \ldots, B_{N-1}$ generated using DISTRIBUTOR from an LTS $M = (S, A, T, s_0)$, and the index i_0 $(= h(s_0))$ of the BCG file containing s_0. BCGMERGE constructs a BCG file that encodes M by numbering the states contiguously from 0 to $|S| - 1$.

```
**procedure** BcgMerge ($B_0$, ..., $B_{N-1}$, $i_0$) **is**
    $c := 0$;
    **forall** $k = 0$ **to** $N-1$ **do**
        $i := (i_0 + k) \ mod \ N$;
        $c_i := c$;
        $c := c + |\{q \in S_i \mid q \ mod \ N = i\}|$
        $c := c + |\{q \in S_i \mid q \ mod \ N = i\}|$
    **end**;
    $Q := \emptyset$; $A := \emptyset$; $R := \emptyset$; $q_0 := 0$;
    **forall** $k = 0$ **to** $N-1$ **do**
        $i := (i_0 + k) \ mod \ N$;
        **forall** $(q, a, q') \in T_i$ **do**
            $Q := Q \cup \{c_{q \ mod \ N} + (q \ div \ N), c_i + (q' \ div \ N)\}$;
            $A := A \cup \{a\}$;
            $R := R \cup \{(c_{q \ mod \ N} + (q \ div \ N), a, c_i + (q' \ div \ N))\}$
        **end**
    **end**
**end**
```

Fig. 2. Merging of a collection of BCG files into a single one

Let $N_i = |\{q \in S_i \mid q \ mod \ N = i\}|$ be the number of states belonging to file B_i, i.e., the states $s \in S$ of the original LTS having $h(s) = i$. The idea is to assign to each BCG file B_i (for i going from i_0 to $(i_0 + N - 1) \ mod \ N$) a new range $[c_i, c_i + N_i - 1]$ of contiguous state numbers such that $c_{i_0} = 0$ and

$c_{(i+1)\ mod\ N} = c_i + N_i$. The final Bcg file $B = (Q, A, R, q_0)$ is then obtained by concatenating the transitions of all Bcg files B_i, each state number $q \in S_i$ corresponding to a state $s \in S$ with $h(s) = i$ being replaced by $c_i + (q\ \mathit{div}\ N)$ (where *div* denotes integer division). Thus, the initial state s_0 will get number $q_0 = c_{i_0} + (i_0\ \mathit{div}\ N) = 0$ and all states will be numbered contiguously.

It is worth noticing that the BcgMerge algorithm processes only one Bcg file B_i at a time and does not require to load in memory the transition relation of B_i. State renumbering is performed on-the-fly, resulting in a low memory consumption, independent from the size of the input Bcg files.

4 Experimental results

We implemented the Distributor and BcgMerge algorithms within the Cadp verification tool set [12] by using the Open/Cæsar [13] and Bcg environments. To ensure maximal portability, the communication primitives of Distributor are built on top of Tcp/Ip using standard Unix sockets. An alternative implementation using the Mpi (*Message Passing Interface*) standard [15] would have been possible; we chose sockets because they are built-in in most operating systems and because the Distributor algorithm was simple enough not to require the higher-level functionalities provided by Mpi.

We experimented Distributor and BcgMerge on three industrial-sized protocols specified in Lotos:

(a) The HAVi protocol [33], standardized by several companies, among which Philips, in order to solve interoperability problems for home audio-video networks. HAVi provides a distributed platform for developing applications on top of home networks containing heterogeneous electronic devices and allowing dynamic plug-and-play changes in the network configuration. We considered a configuration of the HAVi protocol with 2 device control managers (1,039,017 states and 3,371,039 transitions, state size of 80 bytes).

(b) The correct TokenRing leader election protocol [14] for unidirectional ring networks, which is an enhanced version of the protocols proposed by Le Lann [25] and by Chang & Roberts [7]. This TokenRing protocol corrects an error in Le Lann's and Chang & Roberts' protocols, by allowing to designate a unique leader station in presence of various faults of the system, such as message losses and station crashes. We considered a configuration of the TokenRing protocol with 3 stations (12,362,489 states and 45,291,166 transitions, state size of 6 bytes).

(c) The arbitration protocol for the Scsi-2 bus [3], which is designed to provide an efficient peer-to-peer I/O bus for interconnecting computers and peripheral devices (magnetic and optical disks, tapes, printers, etc.). We considered Scsi-2 configurations consisting of a controller device and several disks that accept data transfer requests from the controller. Two versions of the specification have been used: v1, with 5 disks (961,546 states and 5,997,701 transitions, state size of 13 bytes) and v2, with 6 disks (1,202,208 states and 13,817,802 transitions, state size of 15 bytes).

The experiments have been performed on a cluster of 450 MHz, 512 MBytes PCs connected via SCI. Our performance measurements concern three aspects: speedup, partition function, and use of communication buffers.

4.1 Speedup

Figure 3 shows the speedups obtained by generating the LTSs of the aforementioned LOTOS specifications in parallel on a cluster with up to 10 PCs. For the TOKENRING and HAVi protocols, the speedups observed on N machines are given approximately by the formulas $S_N = t_1/t_N = 0.4N$ and $S_N = 0.3N$ (t_k being the execution time on k machines). For the v1 and v2 versions of the SCSI-2 protocol, the speedups obtained are close to ideal.

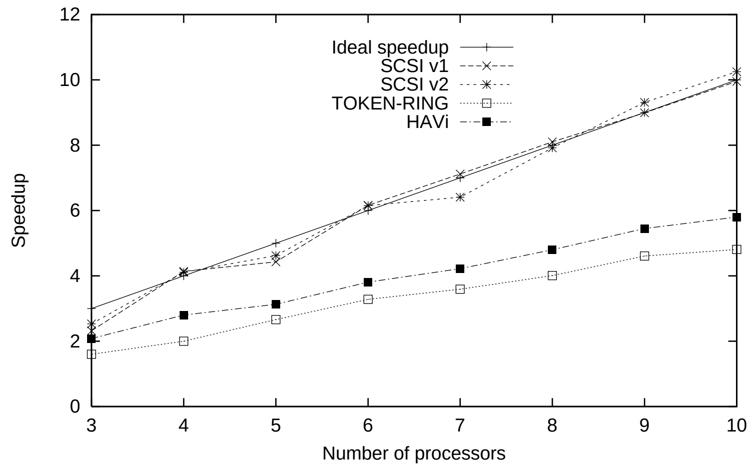


Fig. 3. Speedup measurements for the HAVi, TOKENRING, and SCSI-2 protocols

These results can be explained by examining the implementation of the DISTRIBUTOR algorithm. The state sets explored by each machine in the network are stored locally using the hash tables provided by the OPEN/CÆSAR library. Since the search time in a hash table linearly increases with the number of states present in the table, splitting the state set among N machines is likely to reduce by N the overall search time. Also, parallelization becomes efficient when the time spent in generating state successors is important, which happens for LOTOS specifications having many parallel processes and complex synchronization patterns. This explains why the speedup obtained for the SCSI-2 is better than for the TOKENRING: the SCSI-2 example involves complex data computations (handling of disk buffers and of device status kept by the controller) and synchronizations (multiple rendezvous between 6 or 7 devices to gain bus access), whereas the TokenRing example has very simple computations and only binary synchronizations between stations and communication links.

The speedups obtained show a good overlapping between computations and communications during the execution of Distributor. This is partly due to a buffered communication scheme with well-chosen dimensions of transmission buffers (see Section 4.3).

4.2 Choosing a good partition function

In order to increase the performance of the parallel generation algorithm, it is essential to achieve a good load balancing between the N machines, meaning that the N parts of the distributed Lts should contain (nearly) the same number of states. As indicated in Section 3.1, we adopted a static partition scheme, which avoids the potential communication overhead occurring in dynamic load balancing schemes. Then, the problem is to choose an appropriate partition function $h : S \to [0, N-1]$ associating to each state a machine index.

Because we target at language independent state space construction, we cannot assume that state contents exhibit structural properties (e.g., constant fields, repeated patterns, etc.) particular to a given language. All that we can assume is that state contents are uniformly distributed bit strings.

The Open/Cæsar environment [13] of Cadp offers several hashing functions $f(s, P)$ that compute, for a state s and a prime number P, a hash-code between 0 and $P-1$. The function we chose[1] performs very well, i.e., it uniformly distributes the states of S into P chunks, each one containing $|S|$ *div* P states. To distribute these P chunks on N machines, the simplest way is to take the remainder of the hash-code modulo N, yielding a partition function of the form $h(s) = f(s, P)$ *mod* N. In order to guarantee that h also distributes states uniformly among the N machines, we must choose an appropriate value for P.

Still assuming that state contents are uniformly distributed, the partition function h will allocate $(P \textit{ div } N) + 1$ chunks on each machine $j \in \{0, \ldots, (P \textit{ mod } N) - 1\}$ and P *div* N chunks on each other. If N is prime, the obvious choice for P is $P = N$, leading to a distribution of a single chunk on each machine. If N is not prime, a choice of P such that $P \textit{ mod } N = 1$ ensures that only machine 0 has one chunk more than the others. In this case, P should be sufficiently big, in order to reduce the size $|S|$ *div* P of a chunk. For the experiments presented in this paper, we chose P around 1,600,000 (which gives a limit of 10 states per chunk).

Figures 4 and 5 show the distribution of the states on 10 machines for the main protocols described above. In order to evaluate the quality of the distribution, we calculated the standard deviation $\sigma = \sqrt{(\sum_{i=0}^{N-1}(|S_i| - |S|/N)^2)/N}$ between the sizes $|S_i|$ of the state sets explored by each machine i in the network. For all examples considered, the values obtained for σ are very small (less

[1] This hashing function, called `CAESAR_STATE_3_HASH()` in the Open/Cæsar library, calculates the remainder modulo a prime number of the state vector (seen as an arbitrarily long integer number).

than 1% of the mean value $|S|/N$), which indicates a good performance of the partition function h.

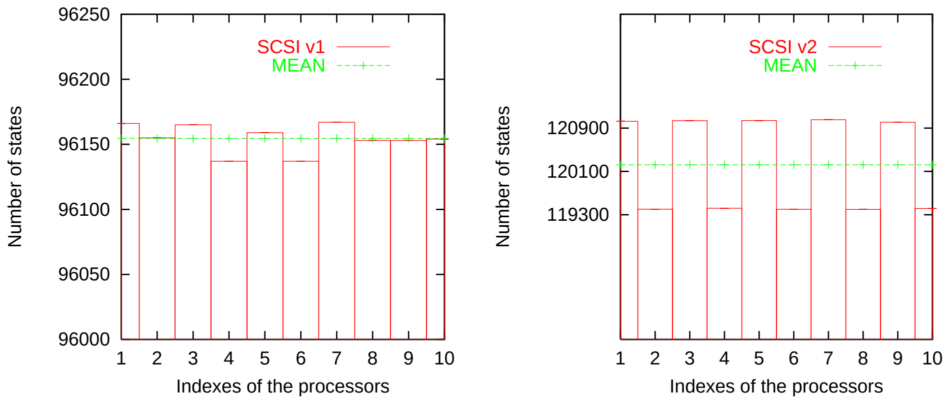


Fig. 4. State distributions for the SCSI-2 protocol on 10 machines

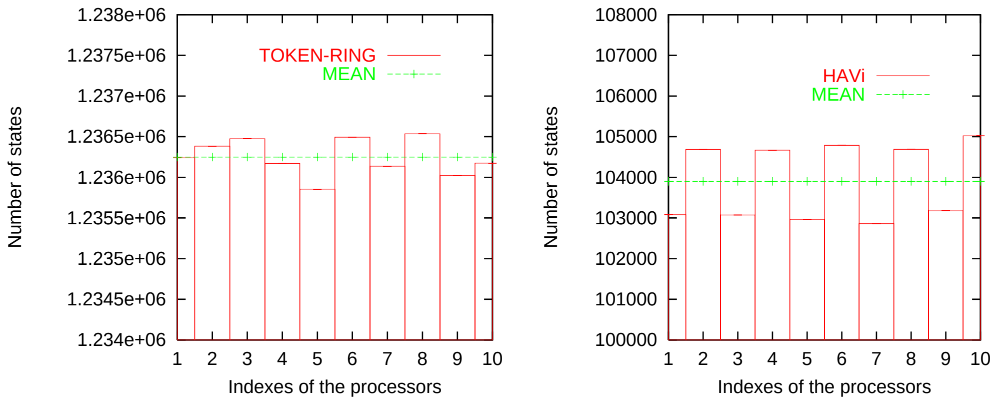


Fig. 5. State distribution for the TOKENRING and HAVi protocols on 10 machines

The quality of a partition function could also be estimated according to the number of “cross-border” transitions of the partitioned LTS (i.e., transitions having the source state in a component and the target state in another component). This number should be as small as possible, since it is equal to the number of *Arc* messages sent over the network during the execution of DISTRIBUTOR. However, in practice, reducing the number of cross-border transitions would require additional information about the structure of the program, and therefore must be language dependent. Since DISTRIBUTOR is built using the language independent OPEN/CÆSAR environment, we did not focused on developing language dependent (e.g., LOTOS-specific) partition functions. This might be done in the future, by extending the OPEN/CÆSAR application programming interface to provide more information about the internal structure of program states.

4.3 Using communication buffers

To reduce the overhead of message transmission and to increase the overlapping between communications and computations, we chose an asynchronous, non-blocking implementation of the SEND and RECEIVE primitives used in the DISTRIBUTOR algorithm. Also, to reduce communication latency, these primitives actually perform a buffering of messages (LTS transitions) instead of sending them one by one as indicated in Figure 1.

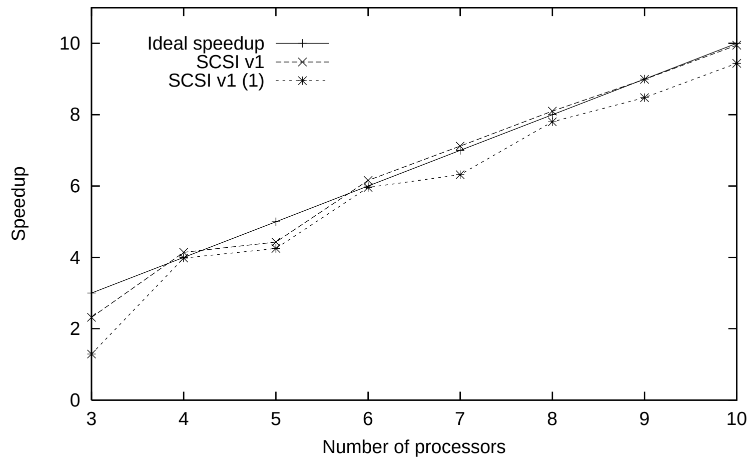
The implementation is based on TCP/IP and standard UNIX communication primitives (sockets). In practice, for each machine $0 \leq i \leq N-1$, there is a virtual channel (i, j) to every other machine $j \neq i$ with a corresponding logical buffer of size L used for storing messages transmitted on the channel. The $N-1$ virtual channels associated with each machine share the same physical channel (socket), which has an associated buffer of size L_p. For a given size d of messages (which depends on the application), we observed that the optimal length of the logical transmission buffer is given by the formula $L_{opt} = L_p/d(N-1)$. Experiments show that for this value, all transitions accumulated in the logical transmission buffers can be sent at the physical level by the next call to SEND.

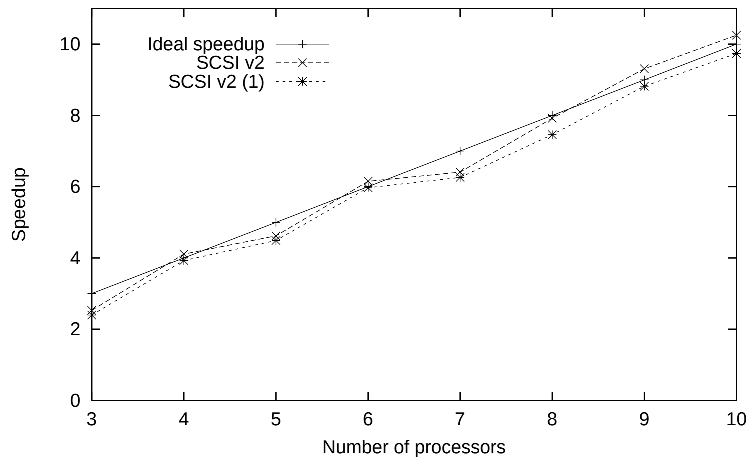
Figure 6 illustrates the effect of buffering on DISTRIBUTOR's speedup for the SCSI-2 and the TOKENRING protocols. A uniform increase of speedup is observed between the variants $L = 1$ (no buffering) and $L = L_{opt}$. The difference in speedup is greater for the TOKENRING protocol because the percentage of communication time w.r.t. computation time is more important than for the SCSI-2 protocol. Therefore, the value L_{opt} seems a good choice for ensuring a maximal overlapping of communications and computations.

5 Conclusion and future work

We presented a solution for constructing an LTS in parallel using N machines connected by a network. Each machine constructs a part of the LTS using the DISTRIBUTOR algorithm, all resulting parts being combined using the BCGMERGE algorithm to form the complete LTS. These algorithms have been implemented within the CADP tool set [12] using the generic environments OPEN/CÆSAR [13] and BCG for implicit and explicit manipulation of LTSs.

Being independent from any specification language is a difference between our approach and other related work. To our knowledge, all published algorithms but [8] are dedicated to a specific low-level formalism (Petri nets, Markov





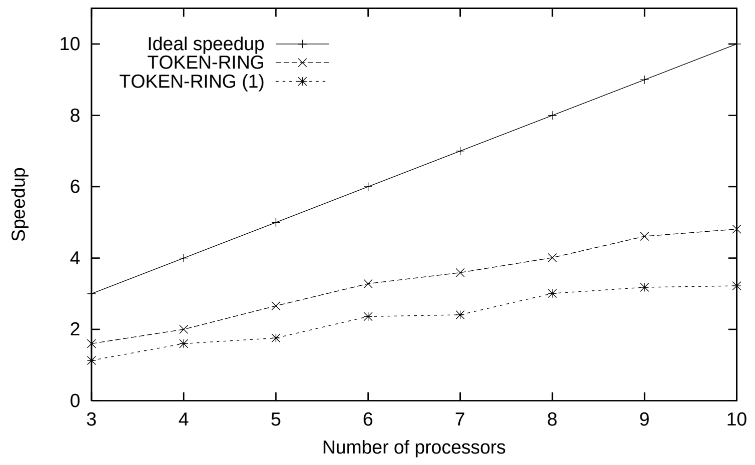


Fig. 6. Speedup measurements for the SCSI-2 and TOKENRING protocols for transmission buffers of size 1 and L_{opt}

chains, etc.) or high-level language (Murφ, Promela, etc.). On the contrary, as the Open/Cæsar and Bcg environments are language independent, the Distributor and BcgMerge tools can be used not only for Lotos, but also for every language having a connection to the Open/Cæsar interface, such as the Umlaut compiler for Uml [22].

Another distinctive feature of our approach relies in the scheme used by Distributor and BcgMerge to assign unique numbers to states. Although the Distributor algorithm is similar to the *ExploreDistributed* algorithm of [8], we manage to number states with mere integers, whereas [8] uses pairs of the form $\langle$*processor number*, *local state number*$\rangle$.

We experimented our approach on several real-size Lotos specifications, for which we generated large Ltss (up to 12 million states and 45 million transitions). Compared to the data reported for other high-level languages such as Murφ [34] and Promela [26], respectively, we were able to generate larger (11 times and 4.2 times, respectively) state spaces.

Moreover, our experimental results show that parallel construction of Ltss provides linear speedups. This is due both to the good quality of the partition function used to distribute the state space among different machines, and to well-dimensioned communication buffers. The speedups obtained are more important for the specifications involving complex data computations and synchronizations, because in this case the traversal of Lts transitions becomes time expensive and can be distributed profitably across different machines.

In this paper, we focused on the problem of constructing Ltss in parallel, with a special emphasis on resource management issues such as state storage in distributed memories and transition storage in distributed filesystems. For a proper separation of concerns, we deliberately avoided to mix parallel state space constructions with other issues such as on-the-fly verification. Obviously, it would be straightforward to enhance the parallel algorithms with on-the-fly verification capabilities such as deadlock detection, invariant checking, or more complex properties. However, this was not suitable to obtain meaningful experimental results (especially, the sizes of the largest state spaces that can be constructed using the parallel approach), because on-the-fly verification may either terminate early without exploring the entire state space, or explore a larger state space when relying on automata product techniques.

This work can be continued in several directions. Firstly, we plan to pursue our experiments and assess the scalability of the approach using a more powerful parallel machine, a cluster of 200 Pcs that is currently under construction at Inria Rhône-Alpes.

Secondly, we plan to extend the Distributor tool in order to handle specifications containing dynamic data structures, such as linked lists, trees, etc. This will require the transmission of variable length, typed data values over a network, contrary to the current implementation of Distributor, which uses messages of fixed length.

Finally, we will seek to determine at which point the sequential verification algorithms available in Cadp (for model-checking of temporal logic formulas on

LTSs, comparison and minimization of LTSs according to equivalence/preorder relations) will give up. As the sizes of LTSs constructed by DISTRIBUTOR will increase, it will be necessary to parallelize the verification algorithms themselves. Two approaches can be foreseen: parallel algorithms operating on-the-fly during the exploration of the LTS, or sequential algorithms working on (already constructed) partitioned LTSs.

Acknowledgements

We are grateful to Xavier Rousset de Pina and to Emmanuel Cecchet for interesting discussions and for providing valuable assistance in using the PC cluster of the SIRAC project of INRIA Rhône-Alpes. We also thank Adrian Curic and Frédéric Lang for their careful reading and comments on this paper.

References

1. S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In *Proceedings of the Parallel Computing Conference PARCO'97 (Bonn, Germany)*. Springer-Verlag, 1997.
2. S. Allmaier, M. Kowarschik, and G. Horton. State Space Construction and Steady-State Solution of GSPNs on a Shared-Memory Multiprocessor. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 112–121. IEEE CS-Press, 1997.
3. ANSI. Small Computer System Interface-2. Standard X3.131-1994, January 1994.
4. J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.
5. S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on SIMD Massively Parallel GSPN Analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794. Lecture Notes in Computer Science, Springer-Verlag, 1994.
6. S. Caselli, G. Conte, and P. Marenzoni. Parallel State Space Exploration for GSPN Models. In G. De Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935, pages 181–200. Lecture Notes in Computer Science, Springer-Verlag, 1995.
7. Ernest Chang and Rosemary Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM*, 22(5):281–283, may 1979.
8. G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 1997.
9. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a New Symbolic Model Checker. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, April 2000.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
11. D. Dill. The Murφ Verification System. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, July 1996.

12. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.
13. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.
14. Hubert Garavel and Laurent Mounier. Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 29(1–2):171–197, July 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Full version available as INRIA Research Report RR-2986.
15. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol. 2 — The MPI-2 Extensions*. MIT Press, 1998.
16. B. Haverkort, H. Bohnenkamp, and A. Bell. On the Efficient Sequential and Distributed Evaluation of Very Large Stochastic Petri Nets. In *Proceedings PNPM'99 (Petri Nets and Performance Models)*. IEEE CS-Press, 1999.
17. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification CAV'2000 (Chicago, IL, USA)*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer Verlag, July 2000.
18. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
19. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
20. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.
21. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.
22. J-M. Jézéquel, W.M. Ho, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an Extendible UML Transformation Framework. In R.J. Hall and E. Tyugu, editors, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering ASE'99*. IEEE, 1999. Also available as INRIA Technical Report RR-3775.
23. W. J. Knottenbelt and P. G. Harrison. Distributed Disk-Based Solution Techniques for Large Markov Models. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains NSMC'99*, Zaragoza, Spain, September 1999.
24. W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger. Probability, Parallelism and the State Space Exploration Problem. In *Proceedings of the 10th International Conference on Modelling, Techniques and Tools (TOOLS '98)*, pages 165–179. LNCS 1469, September 1998.

25. Gérard Le Lann. Distributed Systems — Towards a Formal Approach. In B. Gilchrist, editor, *Information Processing 77*, pages 155–160. IFIP, North-Holland, 1977.
26. F. Lerda and R. Sista. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking SPIN'99*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer Verlag, July 1999.
27. P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models: a Distributed Solution Tool. In *Proceedings of the 7th International Workshop on Petri Nets and Performance Models*, pages 122–131. IEEE Computer Society Press, 1997.
28. Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, April 2000. Also available as INRIA Research Report RR-3899.
29. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.
30. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
31. D. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *Journal of Parallel and Distributed Computing*, 47:153–167, 1997.
32. Y.S. Ramakrishna and S.A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer Verlag, 1997.
33. Judi Romijn. Model Checking the HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, Amsterdam, The Netherlands, June 1999. submitted to Formal Methods in System Design.
34. U. Stern and D. Dill. Parallelizing the Murφ Verifier. In *Computer Aided Verification*, volume 1254, pages 256–267. Lecture Notes in Computer Science, Springer-Verlag, 1997.