Martin Leucker
Jaco van de Pol (Eds.)

Parallel and Distributed Methods in verifiCation

4th International Workshop, PDMC 2005
Lisboa, Portugal, July 2005
Preliminary Proceedings

Sattelite workshop to ICALP 2005 – the 32nd Intl. Colloquium on Automata, Languages and Programming

PDMC'05

The final version will be published in *Electronic Notes in Theoretical Computer Science*
URL: www.elsevier.nl/locate/entcs

Preface

These are the preliminary proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2005) held in Lisboa, Portugal, on July 10, 2005 as a satellite event to the 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005).

The primary mission was to present recent developments in this new research domain and additionally building and strengthening the relationship between people working in the area of parallel and distributed approaches to the verification of large-scale systems.

The workshop program included an invited talk by Gerd Behrmann & Kim G. Larsen on "*Beyond Single Cluster Reachability Analysis*".

Besides regular papers, short presentations describing "work in progress" were accepted to strengthen the workshop atmosphere. In total, 5 regular papers and 4 short presentations have been accepted by the following program committee:

Howard Barringer (Manchester Univ., UK), Lubos Brim (Masaryk Univ., Czech Republic), Gianpiero Cabodi (Torino, Italy), Jörg Denzinger (Alberta, Canada), Wan Fokkink (Free University/CWI Amsterdam, NL), Hubert Garavel (INRIA, France), Jürgen Giesl (RWTH Aachen, Germany) Orna Grumberg (Haifa, Israel), Boudewijn R. Haverkort (Univ. of Twente, NL), Marta Kwiatkowska (Univ. of Birmingham, UK), Martin Leucker (TU Munich, Germany) - Co-chair, Eric Mercer (Brigham Young Univ., USA), Jaco van de Pol (CWI, Amsterdam, NL) - Co-Chair, Gerardo Schneider (Inria Rennes, France), Willem Visser (NASA Ames Research Center, USA)

We would like to thank very much all members of the program committee for nice cooperation and for detailed reports and comments.

These proceedings are preliminary and are intended as a working material for the workshop participants. The final workshop proceedings will be published as a volume of Electronic Notes in Computer Science and will include the regular papers only.

Lisboa, July 10, 2005 — Martin Leucker and Jaco van de Pol

Table of Contents

Invited Talk

Beyond Single Cluster Reachability Analysis
Gerd Behrmann & Kim G. Larsen 6

Regular Papers

How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors
L. Brim, I. Černá, P. Moravec, J. Šimša 7

Distribution, Approximation and Probabilistic Model Checking
G. Guirado, T. Herault, R. Lassaigne, S. Peyronnet 22

Under-approximation heuristics for Grid-based BMC
S. Iyer, J. Jain, D. Sahoo, E. A. Emerson 34

Distributed Symbolic Bounded Property Checking
P. K. Nalla, R. J. Weiss, P. Peranandam, J. Ruf, T. Kropf, W. Rosenstiel 49

A Pattern Recognition Approach for Speculative Firing Prediction in Distributed Saturation State-Space Generation
Ming-Ying Chung, Gianfranco Ciardo 65

Short Papers

DISTRIBUTOR and BCG MERGE: Tools for Distributed Explicit State Space Generation
H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, and G. Stragier 80

DiVine - The Distributed Verification Environment
J. Barnat, L. Brim, I. Černá, P. Šimeček 89

DivSPIN - A SPIN compatible distributed model checker
M. Leucker, M. Weber, V. Forejt, J. Barnat 95

A New Reachability Algorithm for Symmetric Multi-processor Architecture
D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill 101

Beyond Single Cluster Reachability Analysis

Gerd Behrmann

Department of Computer Science, Aalborg University

Kim G. Larsen

Department of Computer Science, Aalborg University

Abstract

Enumerative distributed reachability analysis based on a simple partitioning of the state space has become a well known technique implemented in numerous model checking tools. The timed automata model checker UPPAAL is no exception to this rule, and for over 5 years a distributed version of UPPAAL has been used to analyse large timed automata models. In this talk we address two fundamental issues with the current approach.

Memory usage and CPU usage is tightly coupled. In particular, this limits the freedom to utilize non-homogeneous setups such as mixed clusters or clusters of clusters (e.g. connected by GRID infrastructure). Our proposed solution also solves the load instability we first observed in early versions of distributed UPPAAL and that has since been observed in other distributed model checkers.

Distributed reachability analysis is tightly coupled with breadth first search of the state space. For this reason, extending the algorithm from pure reachability analysis to LTL or TCTL model checking is difficult, as current on-the-fly approaches for timed automata are based on a depth-first search. We present an idea that allows any fixed point computation, including LTL or TCTL model checking, to be easily distributed.

How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors

L. Brim [1], I. Černá [2], P. Moravec, J. Šimša

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

Distributed automata-based LTL model-checking relies on algorithms for finding accepting cycles in a Büchi automaton. The approach to distributed accepting cycle detection as presented in [9] is based on maximal accepting predecessors. The ordering of accepting states (hence the maximality) is one of the main factors affecting the overall complexity of model-checking as an imperfect ordering can enforce numerous re-explorations of the automaton. This paper addresses the problem of finding an optimal ordering, proves its hardness, and gives several heuristics for finding an optimal ordering in the distributed environment. We compare the heuristics both theoretically and experimentally to find out which of these work well.

1 Introduction

Over the past decade, many techniques using distributed and/or parallel processing have been developed to combat the computational complexity of verification problems. They cover reachability analysis [3,14,17,21], verification of branching time logics [4,5,7,8,12,15], linear time logics [1,2,10], equivalence checking [6,18], and other verification problems.

In this paper we concentrate on the technique of maximal accepting predecessor for LTL model-checking as presented in [9]. We show how this technique can be extended and optimised to speed-up LTL model-checking in a distributed environment.

The maximal accepting predecessors (*MAP*) algorithm comes out from the automata approach which reduces the LTL model-checking problem to the emptiness problem for Büchi automata. A Büchi automaton accepts a

[1] Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509

[2] Research supported by the Academy of Sciences of Czech Republic grant No. 1ET408050503

non-empty language if and only if there is a reachable *accepting cycle* in the Büchi automaton graph.

Reachability is a graph exploration technique that can be efficiently parallelised. The *MAP* algorithm exploits reachability for cycle detection in the distributed environment. The algorithm is derived from the observation that all vertices on a cycle have the same set of predecessors. To avoid computing sets of all predecessors the algorithm assigns to every vertex a single representative predecessor. Another core idea of the algorithm is to make use of vertex ordering to determine suitable representatives. Namely, supposing the vertices of the graph are ordered, the representative is the maximal accepting predecessor of the vertex (or null value if there is none). A sufficient condition for a graph to contain an accepting cycle is that there is an accepting vertex with itself as the maximal accepting predecessor. Unfortunately, this is not a necessary condition as there can exist an accepting cycle with "its" maximal accepting predecessor lying outside of it. For this reason the algorithm systematically re-classifies those accepting vertices which do not lie on any cycle as non-accepting and re-computes the maximal accepting predecessors. The overall complexity of the *MAP* algorithm is mainly derived from both computing the representatives and the number of iterations in which vertices are re-classified and the representatives are re-computed. It turns out that the vertex ordering is of crucial importance for improving the performance of the algorithm.

In [9] a few basic vertex orderings have been considered, a systematic exposition of vertex orderings and its impact on the algorithm effectiveness has been left open. In this paper we investigate the influence of the vertex ordering in detail. First of all, we introduce the notion of an *optimal ordering* as the ordering for which the *MAP* algorithm terminates in the very first iteration, i.e. without re-classifying the representatives. The optimal ordering can be computed for example by depth-first search traversal of the graph. However, as we prove, the problem itself is P-complete and its efficient distributed solution is not at hand (Section 3). Therefore, we formulate several heuristics to resolve the ordering problem in a distributed environment and investigate their theoretical properties (Section 4). All heuristics went through a detailed experimental evaluation (Section 5) giving a deeper insight into their practical usability in the distributed verification.

2 Maximal Accepting Predecessors

In this section, we recapitulate the main idea of the *MAP* algorithm as presented in [9], concentrating on the impact of vertex ordering on the complexity of the algorithm.

The *MAP* algorithm follows the automata-based approach to LTL model-checking [22]. The verification problem is reduced to the *emptiness* problem for Büchi automata and is represented as a graph problem. Let $\mathcal{A}$ =

$(\Sigma, S, \delta, s, Acc)$ be a Büchi automaton where Σ is an input alphabet, S is a finite set of states, $\delta : S \times \Sigma \rightarrow 2^S$ is a transition relation, s is an initial state and $Acc \subseteq S$ is a set of accepting states. The automaton $\mathcal{A}$ can be identified with a directed graph $G_{\mathcal{A}} = (V, E, s, A)$, called an *automaton graph*, where $V \subseteq S$ is a set of vertices corresponding to all *reachable states* of the automaton $\mathcal{A}$, $E = \{(u, v) \mid u, v \in V \text{ and } v \in \delta(u, a) \text{ for some } a \in \Sigma\}$, $s \in V$ is a distinguished initial vertex corresponding to the initial state of $\mathcal{A}$ and A is a distinguished set of accepting vertices corresponding to reachable accepting states of $\mathcal{A}$.

Definition 2.1 Let $G = (V, E, s, A)$ be an automaton graph. The *reachability relation* $\rightsquigarrow^+ \subseteq V \times V$ is defined as $u \rightsquigarrow^+ v$ iff there is a directed path $\langle u_0, u_1, \ldots, u_k \rangle$ in G where $u_0 = u$, $u_k = v$ and $k > 0$.

A directed path $\langle u_0, u_1, \ldots, u_k \rangle$ forms a *cycle* if $u_0 = u_k$ and the path contains at least one edge. A cycle is *accepting* if at least one vertex on the path $\langle u_0, u_1, \ldots, u_k \rangle$ belongs to the set of accepting vertices A.

A Büchi automaton recognises a non-empty language iff its automaton graph contains an accepting cycle. The *MAP* algorithm detects accepting cycles by maximal accepting predecessors. It assumes a linear ordering $\prec$ on the set V of vertices. The ordering is extended to the set $V \cup \{null\}$ ($null \notin V$) by setting $null \prec v$ for all $v \in V$.

Definition 2.2 Let $G = (V, E, s, A)$ be an automaton graph. A *maximal accepting predecessor function* of the graph G, $map_G : V \rightarrow (V \cup \{null\})$, is defined as

$$map_G(v) = \begin{cases} \max\{u \in A \mid u \rightsquigarrow^+ v\} & \text{if } \{u \in A \mid u \rightsquigarrow^+ v\} \neq \emptyset \\ null & \text{otherwise} \end{cases}$$

If there is a vertex $v \in V$ with $map_G(v) = v$, the algorithm reports an accepting cycle. However, it can happen that the graph contains an accepting cycle and for all $v \in V$ the inequality $map_G(v) \neq v$ holds. As all vertices on a cycle must have the same maximal accepting predecessor, this can only happen if this predecessor lies outside the cycle. Such a vertex can be removed from the set of accepting vertices without violating the existence of an accepting cycle in the graph. This idea is formalised in the notion of a *deleting transformation*. Whenever the deleting transformation is applied to an automaton graph G with $map_G(v) \neq v$ for all $v \in V$, it shrinks the set of accepting vertices by deleting the vertices which evidently do not lie on any cycle.

Definition 2.3 Let $G = (V, E, s, A)$ be an automaton graph and map_G its maximal accepting predecessor function. A *deleting transformation* is defined as $del(G) = (V, E, s, \overline{A})$, where $\overline{A} = A \setminus \{u \in A \mid map_G(u) \prec u\}$.

Note that the application of the deleting transformation can result in a different *map* function but it preserves the property *"the graph contains an accepting cycle"*. The *MAP* algorithm alternately computes the *map* function

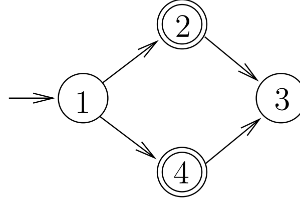


Fig. 1. Deleting transformation

and applies the deleting transformation till an accepting cycle is discovered or the set of accepting states is empty.

MAP Algorithm
<u>while</u> $A \neq \emptyset$ **<u>do</u>**
compute map_G;
<u>if</u> $(\exists u \in A : map_G(u) = u)$
<u>then</u> return CYCLE
<u>else</u> $G = del(G)$;
<u>fi</u>
<u>od</u>
return NO CYCLE

In our original algorithm [9] the deleting transformation has been defined using the set $\{u \in A \mid \exists v \in V.map_G(v) = u\}$ of accepting vertices to be removed. The new formulation of the deleting transformation used here is more appropriate in the context of optimising vertex ordering as it generally removes more vertices. E.g. consider the graph on Figure 2 with two accepting vertices 2 and 4 and the vertex ordering given by their numbers. The algorithm terminates in two iterations under the original definition (in the first iteration the vertex 4 is deleted, in the second one the vertex 2 is deleted) while it needs only one iteration to terminate under the new definition (both accepting vertices are deleted at once as $map_G(2) = null \prec 2$ and $map_G(4) = null \prec 4$). The correctness of the modified algorithm can be easily proved following similar arguments as given in [9].

3 Optimal Vertex Ordering for the *MAP* Algorithm

The time complexity of the distributed *MAP* algorithm is $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices and m is the number of edges in the automaton graph. Here the factor $a \cdot m$ comes from the computation of the *map* function and the factor a relates to the number of iterations, i.e., computations of the *del* function. In order to optimise the complexity one aims to decrease the number of iterations by choosing an appropriate vertex ordering. A natural way how to order the vertices is to use the enumeration order as it is computed in the enumerative on-the-fly model-checking. In [9], each vertex was identified with a vector of three numbers – the workstation identifier, the row number in the hash table, and the column number in the row. The ordering of vertices

was given by the lexicographical ordering of these triples. In this section, we define the notion of an optimal ordering and prove that the optimal ordering problem is P-complete.

Let $\prec$ be a linear ordering on vertices used by the algorithm *MAP* and $iter_\prec$ be the number of iterations of the main cycle till the algorithm *MAP* terminates.

Definition 3.1 An ordering $\prec$ is *optimal* iff $iter_\prec = 1$.

The optimality of an ordering is tightly related to a reachability relation on the set of accepting vertices.

Definition 3.2 An ordering $\prec$ *respects reachability* iff for all $u, v \in A$, whenever $(u \leadsto^+ v \wedge v \not\leadsto^+ u)$ then $u \prec v$.

Lemma 3.3 *If an ordering $\prec$ respects reachability then it is optimal.*

Proof. We prove that non-optimal ordering does not respect reachability.

Suppose the ordering $\prec$ is not optimal and there is an accepting cycle in the graph G. The algorithm does not detect an accepting cycle in the first iteration if for all accepting vertices u the value $map_G(u) \neq u$. Let v be the maximal accepting vertex lying on a cycle. Then $v \prec map_G(v)$, $map_G(v) \leadsto^+ v$, and $v \not\leadsto^+ map_G(v)$. Therefore $\prec$ does not respect reachability.

If there is no accepting cycle in the graph, then there is an accepting vertex v which is not re-classified as non-accepting after the first iteration of the *MAP* algorithm. It means that $v \prec map_G(v)$ and $map_G(v) \leadsto^+ v$. From acyclicity we have $v \not\leadsto^+ map_G(v)$, which implies that $\prec$ does not respect reachability.

Lemma 3.4 *For every automaton graph there is an optimal ordering. Moreover, an optimal ordering can be computed in time $\mathcal{O}(a \cdot m)$.*

Proof. We give algorithm which computes an optimal ordering. As a first step, the algorithm computes the reachability relation $R = \{(u, v) \mid u, v \in A, u \leadsto^+ v\}$. This computation can be done for example by running a reachability procedure from all accepting vertices separately which takes time $\mathcal{O}(a \cdot m)$.

Now, if the graph does not contain any accepting cycle, then for $u, v \in A$ we put $u \prec v$ if and only if $(u, v) \in R$. Other pairs of vertices are ordered arbitrarily. If the graph contains an accepting cycle, then there is a vertex u with $(u, u) \in R$. Let $v \prec u$ for every accepting vertex v, $v \neq u$. Other pairs of vertices are again ordered arbitrarily.

Notice, that a graph can have several optimal orderings, as the ordering of non-accepting vertices and of accepting vertices, which are mutually unreachable, is not important.

The question is whether an optimal ordering can be computed more efficiently in the distributed environment. We provide a strong evidence that the computation of an optimal ordering cannot be significantly speeded up by the

use of any reasonable number of parallel processors. Namely, we prove that the *optimal ordering problem* is P-complete and thus inherently sequential. A problem is P-complete if it belongs to P and every language $L \in P$ is log-space reducible to the problem (see [13] for details on P-completeness).

The optimal ordering problem is to decide for a given automaton graph and two accepting vertices u, v whether u precedes v in *every* optimal ordering of graph vertices. Lemma 3.4 shows that the optimal ordering problem is in P. We prove P-hardness by reduction from the NAND circuit value problem.

A *NAND boolean circuit* is a sequence $B = (B_0, \ldots, B_n)$ where $B_0 = 1$ and $B_i = \neg(B_{i_1} \vee B_{i_2})$, $i_1, i_2 < i$. Let $value(B_0) = true$, $value(B_i) = \neg(value(B_{i_1}) \vee value(B_{i_2}))$, and $value(B) = value(B_n)$. The *NAND circuit value (NANDCV) problem* is to decide for a given NAND boolean circuit B whether $value(B) = true$. Ladner [16] shows that the NANDCV problem is P-complete.

Theorem 3.5 *The optimal ordering problem is P-hard.*

Proof. By log-space reduction of the NANDCV problem to the optimal ordering problem. Let $B = (B_0, \ldots, B_n)$ be a NAND boolean circuit. We construct an automaton graph G and identify its two vertices u, v in such a way that u precedes v in every optimal ordering of graph vertices if and only if $value(B) = true$.

First, for each B_i we construct a graph G_i inductively. The graph $G_0 = (\{T_0, I_0, F_0\}, \{(T_0, I_0), (I_0, F_0), (F_0, I_0)\})$ is depicted in Figure 2a). Let $B_i = \neg(B_{i_1} \vee B_{i_2})$. Then G_i contains as its subgraphs G_{i_1} and G_{i_2}, new vertices T_i, I_i, F_i, and new edges as depicted in Figure 2b).

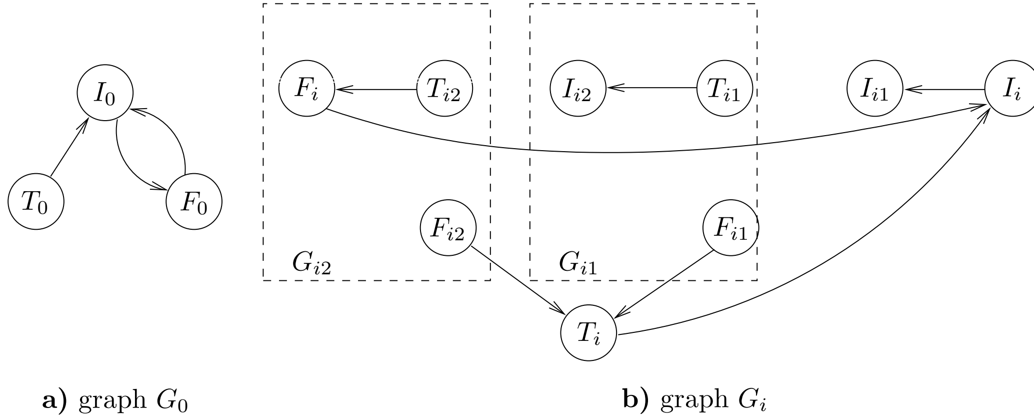


Fig. 2. Construction of the automaton graph

We prove that for all $i = 0, \ldots, n$ the graph G_i has specific reachability properties. Namely,

if $value(B_i) = true$ then $T_i \rightsquigarrow^+ I_i \rightsquigarrow^+ F_i$, $F_i \rightsquigarrow^+ I_i$, $I_i \not\rightsquigarrow^+ T_i$, and $F_i \not\rightsquigarrow^+ T_i$,

if $value(B_i) = false$ then $F_i \rightsquigarrow^+ I_i \rightsquigarrow^+ T_i$, $T_i \rightsquigarrow^+ I_i$, $I_i \not\rightsquigarrow^+ F_i$, and $T_i \not\rightsquigarrow^+ F_i$.

The assertion can be proved by induction on i. For $i = 0$, $value(B_0) = true$ and the assertion can be easily checked following Figure 2a).

For the induction step let us suppose $value(B_i) = true$. Then $value(B_{i_1}) = value(B_{i_2}) = false$ and by induction hypothesis there are paths from I_{i_1} to T_{i_1} and from I_{i_2} to T_{i_2}. These paths together with edges (T_i, I_i), (I_i, I_{i_1}), (T_{i_1}, I_{i_2}), and (T_{i_2}, F_i) form a path from T_i to F_i in G_i. On the other hand, as there is no path from I_{i_1} to F_{i_1} in G_{i_1} neither from I_{i_2} to F_{i_2} in G_{i_2}, there is no path both from I_i and F_i to T_i in G_i.

The case $value(B_i) = false$ divides into three subcases depending on values of $value(B_{i_1})$ and $value(B_{i_2})$, all subcases are handled analogously to the previous case.

To finish the proof of P-hardness of the optimal ordering problem, let us reduce the NAND boolean circuit B to the automaton graph G containing as its subgraph G_n, a new initial vertex S and edges from S to all vertices in G_n. Vertices T_n and F_n are accepting. From properties of G_n we have that if $value(B) = true$ then $T_n \rightsquigarrow^+ F_n \wedge F_n \not\rightsquigarrow^+ T_n$ and if $value(B) = false$ then $F_n \rightsquigarrow^+ T_n \wedge T_n \not\rightsquigarrow^+ F_n$. We claim that $value(B) = true$ iff in every optimal ordering T_n precedes F_n. Clearly, if $value(B) = true$ and F_n preceded T_n, then $map(T_n) = null$, $map(F_n) = T_n$, and the MAP algorithm would need two iterations to complete the cycle detection. For the opposite implication, if $value(B) = false$, then ordering in which F_n precedes T_n is optimal as $map(F_n) = null$ and $map(T_n) = F_n$. To conclude the proof we observe that the construction of the graph G can be done in space logarithmic with respect to the circuit size.

4 Heuristics for vertex ordering

As the optimal ordering problem is P-complete, we cannot expect the computation of an optimal ordering in the distributed environment to be significantly more efficient than in the sequential setting. Therefore we aim for non-optimal orderings. In this section, we describe several heuristics for computing a vertex ordering. All but one are easily computable in the distributed environment. For all orderings we indicate how "far" is the computed ordering from the optimal one. We elaborate a quantitative measure that characterizes the distance.

Definition 4.1 Let $\prec$ be an ordering and $\gamma = \langle u_1, \ldots, u_n \rangle$ be a path in G. Then $(u_{i_1}, \ldots, u_{i_k})$ is a *reverse subsequence* of the sequence $(u_1, \ldots, u_n)$ if $u_{i_1}, \ldots, u_{i_k}$ are accepting vertices and $u_{i_k} \prec \ldots \prec u_{i_2} \prec u_{i_1}$. The maximal length of a reverse subsequence of the path γ is the *index of the path* γ, $index_\prec(\gamma)$.

Index of a vertex u is defined as $index_\prec(u) = \max\{index_\prec(\gamma) \mid \gamma$ is a path from the initial vertex to the vertex u in $G\}$.

Index of an automaton graph G is defined as $index_\prec(G) = \max\{index_\prec(u) \mid u$ is a vertex in $G\}$.

To illustrate the definition, let $\gamma = \langle 4, 2, 3, 5, 1 \rangle$ be the path depicted on Figure 3 and $1 \prec 2 \prec 3 \prec 4 \prec 5$. Then $(4, 2)$, $(4, 3)$, $(4, 3, 1)$, and $(3, 1)$ are reverse subsequences of the sequence $(4, 2, 3, 5, 1)$. On the other hand, the sequences $(4, 2, 3, 1)$ and $(5, 1)$ are not reverse subsequences of γ. Index of the path γ is 3.

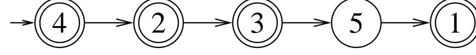


Fig. 3. Path with reverse subsequence $(4, 3, 1)$

Theorem 4.2 *For a graph G and a vertex ordering $\prec$, $iter_\prec = index_\prec(G)$.*

Proof. To prove the inequality $index_\prec(G) \leq iter_\prec$ let us assume there is a vertex u with $index_\prec(u) > iter_\prec$. Let $\sigma = (u_1, \dots u_k)$ be a reverse subsequence of a path from s to u with $|\sigma| = index_\prec(u)$. Then at least two vertices u_i, u_j $(i < j)$ have to be deleted from A during the same deleting transformation. But $u_i \rightsquigarrow^+ u_j$, $u_j \prec u_i$ and therefore $u_j \prec map(u_j)$. This contradicts the definition of the deleting transformation.

For the opposite inequality $index_\prec(G) \geq iter_\prec$, let u be a vertex and $\gamma = \langle s, \dots u \rangle$ be a path such that $index_\prec(\gamma) = index_\prec(u) = index_\prec(G) = k$. Let $\sigma = (u_1, u_2, \dots u_k)$ be the reverse subsequence of the maximal length of the path γ. By induction on the index i we prove that the vertex u_i is removed from the set of accepting vertices during the ith iteration of the algorithm *MAP*.

For $i = 1$ the assertion follows from the maximality of γ. For the induction step assume that the vertex u_{i-1} was removed during the $(i-1)$th iteration. If u_i is not removed from the set of accepting vertices during the ith iteration then there is a vertex $v_i \in A$ with $s \rightsquigarrow^+ v_i \rightsquigarrow^+ u_i$ and $u_i \prec v_i$ (i.e. in the ith iteration $u_i \prec map(u_i)$). The vertex v_i is re-classified as non-accepting not sooner than during the ith iteration and we can repeat similar arguments for the vertex v_i. As a result we have vertices $u_i \prec v_i \prec v_{i-1} \prec \dots v_1$ with $s \rightsquigarrow^+ v_1 \rightsquigarrow^+ \dots v_i \rightsquigarrow^+ u_i$. Hence $(v_1, v_2, \dots v_i, u_i, \dots u_k)$ is a reverse subsequence with $k+1$ vertices of a path from s to u. This contradicts the maximality of γ and σ.

Now we define several vertex orderings which are based on different ways of graph traversal. All but the first one are envisaged to be appropriate for the distribution.

Definition 4.3 Let G be an automaton graph.

$\prec_{DFS}$: Suppose the graph G is traversed by depth first search (DFS). We define $u \prec_{DFS} v$ iff the vertex u is backtracked by DFS *later* than the vertex v (i.e., $\prec_{DFS}$ is the reverse of DFS-postorder).

$\prec_{BFS}$: Suppose the graph G is traversed by breadth first search (BFS). We define $u \prec_{BFS} v$ iff the vertex u is visited by BFS *before* the vertex v.

$\prec_{BFSpreds}$: Suppose the graph G is traversed by BFS. Let G' be the breadth first search tree. Let $visit(u) = (acc_preds, BFS_{nr})$, where acc_preds is the number of accepting predecessors of the vertex u in G' and BFS_{nr} is the time when the vertex u is visited by BFS. We define $u \prec_{BFSpreds} v$ iff $visit(u)$ is lexicographically smaller than $visit(v)$.

The difference between $\prec_{BFSpreds}$ and $\prec_{BFS}$ is shown in Figure 4. In both graphs the successors of the initial vertex are proceeded from left to right. For the left hand side graph $iter_{\prec_{BFSpreds}} = 2$ and $iter_{\prec_{BFS}} = 1$ (since $a \prec_{BFS} b$, but $b \prec_{BFSpreds} a$) while for the right hand side graph $iter_{\prec_{BFSpreds}} = 1$ and $iter_{\prec_{BFS}} = 2$ (since $d \prec_{BFSpreds} c$, but $c \prec_{BFS} d$).

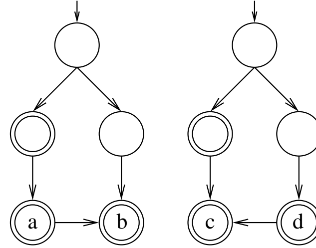


Fig. 4. Comparison of $\prec_{BFSpreds}$ and $\prec_{BFS}$

For the next ordering suppose the graph G is divided into subgraphs G_1, G_2,..., G_n. Further suppose G is traversed by a modified depth first search (cDFS) which differs from DFS in traversing cross edges (edges with vertices from distinct subgraphs). For each subgraph, cDFS maintains a queue of vertices from which it starts a local DFS. A local DFS traverses only the respective subgraph. When a cross edge is encountered, its endpoint is enqueued to the respective queue and the search backtracks. cDFS is initiated with a local DFS from an initial vertex and terminates when no local DFS is running and all queues are empty. A straightforward way to distribute the computation of cDFS is to place subgraphs G_1, G_2,..., G_n on different computers and run local DFSs in parallel.

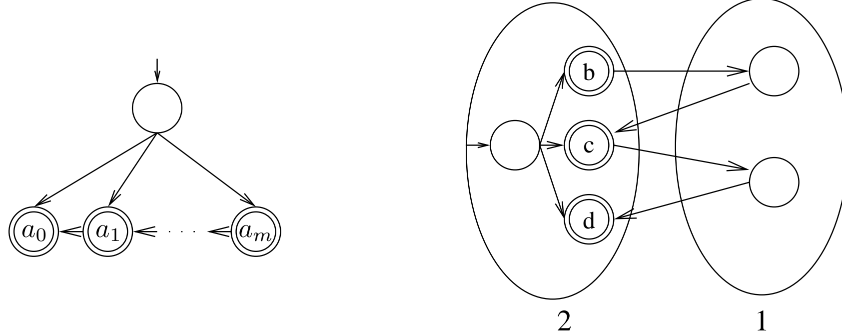
$\prec_{cDFS}$: Suppose the graph G is traversed by cDFS. For $u \in G_i$, $v \in G_j$ we define $u \prec_{cDFS} v$ iff $i < j$ or ($i = j$ and u is backtracked later than v).

Lemma 4.4 *$\prec_{DFS}$ is an optimal ordering, i.e., $index_{\prec_{DFS}}(G) = 1$.*

Proof. According to Lemma 3.3 it suffices to prove that $\prec_{DFS}$ respects the reachability relation. Let $u, v \in A$, $u \rightsquigarrow^+ v$ and $v \not\rightsquigarrow^+ u$. If u is visited by DFS before v, then u is backtracked after all its successors and therefore $u \prec_{DFS} v$. If u is visited later than v, then v must have been backtracked before u was reached, because there is no path from v to u. Hence $u \prec_{DFS} v$. The optimality of $\prec_{DFS}$ corresponds with P-completeness of the DFS problem [20].

Lemma 4.5 *For each $\prec \in \{\prec_{BFS}, \prec_{BFSpreds}, \prec_{cDFS}\}$ there is an automaton graph G such that $index_{\prec}(G) = |A|$.*

Proof. Graph certificating the upper bound for $\prec_{BFS}$ and $\prec_{BFSpreds}$ is depicted in Figure 5a) (successors of the initial vertex are traversed from left to right, $a_0 \prec_{BFS} a_1 \prec_{BFS} \dots a_m$ and $a_0 \prec_{BFSpreds} a_1 \prec_{BFSpreds} \dots a_m$) and for $\prec_{cDFS}$ in Figure 5b) (successors of the initial vertex are traversed bottom up, $d \prec_{cDFS} c \prec_{cDFS} b$).



a) upper bounds for $\prec_{BFS}$ and $\prec_{BFSpreds}$ b) upper bound for $\prec_{cDFS}$

Fig. 5. Upper bounds

5 Experiments

We have implemented variants of the *MAP* algorithm using vertex orderings described in the previous section. The experiments have been performed on a network of ten Intel Pentium 4 2.6 GHz workstations with 1 GB of RAM each interconnected with a 100Mbps Fast Ethernet and using tools provided by our own distributed verification environment DiVinE [11].

Name	**Vertices**	**Acc. Vertices**	**Error**
Elevator10_1	891372	307692	NO
LookUpProc8_2	1954569	1458848	NO
PublicSubscribe_1	2051215	204612	NO
Rether10_4	11325003	5649118	NO
Rether08_2	2898644	850689	YES
PLCshedule600_1	5096287	3827319	YES
Lifts4_1	998570	331596	NO
Phils14L_3	7193116	2410147	NO
TrainGate8_2	17572372	11668232	YES
Peterson3Err_1	1135804	796734	YES

Table 1
Summary of graphs

TrainGate8_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	89	69	45	24	10
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	116	67	34	23	16
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	77	65	26	20	14
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	–	–	1417	855	744
	Iter.	–	–	1	1	1

PLCshedule600_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	9	109	45	62	13
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	7	9	3	14	18
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	3	3	2	3	3
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	–	820	588	450	242
	Iter.	–	1	1	1	1

Peterson3Err_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	81	127	52	70	65
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	167	387	246	216	165
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	116	213	114	98	72
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	141	162	219	129	114
	Iter.	1	1	1	1	1

Rether08_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	86	70	32	40	31
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFS}}$	Time	465	285	146	158	93
	Iter.	1	1	1	1	1
$\prec_{\mathbf{BFSpreds}}$	Time	342	136	88	131	95
	Iter.	1	1	1	1	1
$\prec_{\mathbf{cDFS}}$	Time	465	281	232	186	129
	Iter.	1	1	1	1	1

Table 2
Experimental results: Graphs containing accepting cycles

In order to examine the performance of the algorithm, we performed an extensive experimental evaluation using graphs representing various verification problems. The graphs are identified in Table 1 along with their most important characteristics – the number of reachable vertices and the number of reachable accepting vertices. The column *Error* indicates the presence or absence of an accepting cycle in the graph. Most of the graphs could not be stored on a single computer.

We compared vertex orderings $\prec_{BFS}$, $\prec_{BFSpreds}$, and $\prec_{cDFS}$. Moreover, there are several natural vertex orderings derived from the random hash function used for storing states (see [9] for more details). We used the best one from [9], denoted $\prec_{RND}$, as a “benchmark” for the comparison with newly presented orderings.

Detailed results of all experiments are reported in Tables 2 and 3. For every graph and every ordering we performed experiments on various numbers of workstations. For each setup we give the number of iterations performed by the algorithm and its run time in seconds. The run time represents an average taken from several runs. The sign ’–’ means that the setup resulted

Elevator10_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	295	193	167	153	119
	Iter.	14	14	14	14	14
$\prec_{\mathbf{BFS}}$	Time	296	265	346	382	208
	Iter.	5	7	8	10	8
$\prec_{\mathbf{BFSpreds}}$	Time	159	130	119	117	90
	Iter.	3	4	4	4	4
$\prec_{\mathbf{cDFS}}$	Time	841	530	637	294	294
	Iter.	33	48	49	33	48

PublicSubscribe_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	152	92	67	66	50
	Iter.	8	8	8	8	8
$\prec_{\mathbf{BFS}}$	Time	159	97	72	56	52
	Iter.	4	6	6	6	6
$\prec_{\mathbf{BFSpreds}}$	Time	152	91	67	64	52
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	336	195	285	195	142
	Iter.	7	8	8	8	8

Lifts4_1		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	225	112	76	67	60
	Iter.	12	10	8	8	10
$\prec_{\mathbf{BFS}}$	Time	227	121	91	73	60
	Iter.	3	4	4	4	3
$\prec_{\mathbf{BFSpreds}}$	Time	299	242	190	121	105
	Iter.	4	4	5	5	4
$\prec_{\mathbf{cDFS}}$	Time	397	225	360	216	151
	Iter.	11	21	26	26	28

Lup8_2		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	714	678	266	245	196
	Iter.	12	12	12	12	12
$\prec_{\mathbf{BFS}}$	Time	1640	866	547	508	365
	Iter.	5	7	9	8	9
$\prec_{\mathbf{BFSpreds}}$	Time	427	293	185	167	129
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	1780	1038	1354	995	690
	Iter.	34	41	38	48	45

Phils14L_3		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	2718	1983	2220	3269	2709
	Iter.	11	11	11	11	11
$\prec_{\mathbf{BFS}}$	Time	3444	2606	4812	1935	2813
	Iter.	4	4	9	6	8
$\prec_{\mathbf{BFSpreds}}$	Time	3430	1735	2597	1427	1226
	Iter.	3	3	3	3	3
$\prec_{\mathbf{cDFS}}$	Time	-	6237	6304	5635	5121
	Iter.	-	13	12	12	14

Rether10_4		2	4	6	8	10
$\prec_{\mathbf{RND}}$	Time	–	–	–	1130	722
	Iter.	–	–	–	20	20
$\prec_{\mathbf{BFS}}$	Time	–	–	–	1390	945
	Iter.	–	–	–	10	11
$\prec_{\mathbf{BFSpreds}}$	Time	–	–	–	594	406
	Iter.	–	–	–	4	5
$\prec_{\mathbf{cDFS}}$	Time	–	–	–	5692	15278
	Iter.	–	–	–	165	171

Table 3
Experimental results: Graphs without accepting cycles

in a computation which does not finish due to memory limitations.

In the case of graphs with an accepting cycle, all computations performed only one iteration. In other words, an optimal ordering was found immediately. Although this may seem strange from a theoretical point of view, there is some experimental evidence for this. The number of iterations is bounded by the *quotient graph height*. The quotient graph of $G = (V, E)$ is a graph (W, H),

such that W is the set of strongly connected components of G and $(C_1, C_2) \in H$ if and only if $C_1 \neq C_2$ and there exist $r \in C_1, s \in C_2$ such that $(r, s) \in E$. The *height* of the quotient graph is the length of the longest path in the quotient graph. As argued in [19], the quotient graph height is for model checking graphs typically low and thus the *MAP* algorithm tends to have only a few iterations. In the presence of an accepting cycle, the number of iterations is typically just one.

Furthermore, in some cases you can notice that a computation on fewer workstations takes less time than a computation on more workstations. These irregularities are caused by the hash function used for partitioning and are not related to the algorithm's behaviour.

Yet another observation drawn from the experiments is that in some cases the number of iterations necessary to finish the computation is quite different under different orderings, but the resulting times are very close. This is caused by the uneven number of re-explorations during one iteration. However, lower number of iterations generally results in a faster computation.

As for the orderings, though $\prec_{BFS}$ and $\prec_{BFSpreds}$ are both based on the BFS traversal, $\prec_{BFSpreds}$ outperformed $\prec_{BFS}$ in most experiments. In fact, our experiments suggest the $\prec_{BFSpreds}$ ordering to be the best one among the compared orderings.

The $\prec_{cDFS}$ ordering can be considered from the theoretical point of view as a promising one, as it tries to mimic the optimal $\prec_{DFS}$ ordering. However, it fails to scale well. The high number of iterations is caused by the direct influence of graph distribution on vertex ordering and by the high number of cross edges in the distributed graph. Due to these reasons is the positive impact of distribution dampened.

The random ordering $\prec_{RND}$ can be classified as a "better average". It is interesting to note that despite its randomness it sometimes outperforms orderings which have been designed to employ specific graph features.

Finally, for the $\prec_{BFS}$, $\prec_{BFSpreds}$ and $\prec_{RND}$ orderings the algorithm works on-the-fly as it simultaneously computes the *map* function and performs cycle detection. The experiments clearly demonstrated that in the presence of an accepting cycle, the algorithm was able to detect it during the first iteration. Thus it was not necessary to generate the whole graph. For graphs without accepting cycles the number of workstations had typically small impact on the number of iterations (except for the ordering $\prec_{cDFS}$).

6 Conclusions

The paper complements the distributed LTL model-checking algorithm *MAP* arising out from the maximal accepting predecessors concept. First, we prove that for every graph there is an optimal ordering of graph vertices for which the *MAP* algorithm terminates in one iteration. The optimal ordering can be computed in time linear to the size of the graph, however the problem itself

is P-complete and thus hard to parallelise. Therefore we provide and evaluate several heuristics computing a vertex ordering on-the-fly and such that they are easy to incorporate into the distributed *MAP* algorithm.

Conclusions both from theoretical and experimental evaluation are that none of the heuristics outperforms the others. On average, the most reliable heuristic is $\prec_{BFSpreds}$ (based on breadth first search) followed by $\prec_{RND}$ based on (random) hashing.

The presented approach to the optimisation of the time complexity of the *MAP* algorithms aims at decreasing the number of iterations of the algorithm. An alternative direction is to optimise the computation of the *map* function in each iteration. This computation is based on the relaxation of graph edges (in the same way as in the Belmann-Ford algorithm) and we do not find this too promising.

References

[1] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society Press, 2003.

[2] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[3] G. Behrmann. Distributed Reachability Analysis in Timed Automata. *Software Tools for Technology Transfer*, 7(1):19–30, 2005.

[4] A. Bell and B. R. Haverkort. Sequential and Distributed Model Checking of Petri Net Specifications. In *Parallel and Distributed Model-Checking (PDMC'02)*, volume 68.4 of *ENTCS*. Elsevier, 2002.

[5] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 390–404. Springer, 2000.

[6] S. Blom and S. Orzan. Distributed state space minimization. In *Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *ENTCS*. Elsevier, 2003.

[7] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free μ-Calculus. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[8] B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation-Free mu-Calculus. In *SPIN Workshop on Model checking of Software (SPIN'02)*, volume 2318 of *LNCS*, pages 128–147. Springer, 2002.

[9] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

[10] I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection. In *SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[11] DiVinE – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[12] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[13] R. Greenlaw, H. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[14] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the Efficient Sequential and Distributed Generation of Very Large Markov Chains from Stochastic Petri Nets. In *Petri Nets and Performance Models (PNPM'99)*, pages 12–21. IEEE Computer Society Press, 1999.

[15] T. Heyman, O. Grumberg, and A. Schuster. A Work-Efficient Distributed Algorithm for Reachability Analysis. In *Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 54–66. Springer, 2003.

[16] R. E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[17] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

[18] S. Orzan. *On Distributed Verification and Verified Distribution.* PhD thesis, Free University of Amsterdam, 2004.

[19] R. Pelánek. Typical structural properties of state spaces. In *SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[20] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[21] U. Stern and D. L. Dill. Parallelizing the Murφ Verifier. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.

[22] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press, 1986.

Distribution, Approximation and Probabilistic Model Checking

Guillaume Guirado

EPITA Research and Development Laboratory (LRDE)

Thomas Herault

LRI, University Paris South-XI

Richard Lassaigne

Equipe de Logique Mathématique, UMR 7056 CNRS, University Paris VII

Sylvain Peyronnet

EPITA Research and Development Laboratory (LRDE)
syp@lrde.epita.fr

Abstract

APMC is a model checker dedicated to the quantitative verification of fully probabilistic systems against LTL formulas. Using a Monte-Carlo method in order to efficiently approximate the verification of probabilistic specifications, it could be used naturally in a distributed framework. We present here the tool and its distribution scheme, together with extensive performance evaluation, showing the scalability of the method, even on clusters containing 500+ heterogeneous workstations.

1 Introduction

Probabilistic model checking is an algorithmic method that aims to automatically verify that quantitative properties hold in probabilistic systems. The main drawback of the method is the so-called state space explosion phenomenon, that is the fact that workstations run out of memory while verifying large probabilistic systems. A common direction of research to address this problem is to design distributed model checking algorithms in order to handle larger systems. Most of these methods are about the distribution of the state space on several machines.

In the last couple of years, we showed that a completely different approach can be used in order to save space while verifying large systems. Indeed, we

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

proposed to use approximation to eliminate the space complexity of probabilistic model checking. The idea of using approximation becomes more and more popular and is now used by several research groups [16,4]. Our approach [5] is more precisely based on the sampling of execution paths of the probabilistic system. This method is, by construction, massively parallel. Indeed, one can distribute the computation on a large cluster of machines in the following way: each machine generates execution paths and verifies the specification on each of these paths, then sends the obtained results to a master. After a certain time of computation, the master has received enough results to conclude on the (approximate) validity of the quantitative property to be checked.

In this paper, we explain in detail the method we developed and we analyze the performances of our methodology on very large clusters of heterogeneous machines (up to 500 machines). All the experiments were done using APMC (Approximate probabilistic Model Checker), which is the tool that implements our method.

The paper starts with a short review of the related work. Then, in section 3, we give the theoretical foundations of APMC and explain its architecture and implementation. Last, we present in section 4 the results of extensive experiments on various case studies and sets of machines. These experiments show the scalability of the approach.

2 Related Work

In the last few years, distributed model checking has gained a renewal of interest, due to the emergence of easily available "computing farms", that is very large sets of machines usable for computation. There is now a challenge of using such clusters in every domain of computer science. Several methods have been developed in order to speed up the model checking and/or avoid the state space explosion phenomenon.

One of the first ideas in the use of parallelization was to distribute the construction of the state-space (see for example [13,3]). Basically this is done using a partition of the set of the reachable states by way of a hashing function, this partition induces a natural parallelization.

Concerning the manipulation of the state space, an other way of research is to improve the size of the transitions systems that can be handled by the model checker. Out of core methods were designed to do this [10,9], particularly for probabilistic systems (that is Markov models).

The main convenience of these techniques is their potential to integrate within classical model checkers, such as the state of the art probabilistic model checker, PRISM [11].

A lot of others methods have been developed and discussed [7], but none of them distribute the whole process of the verification in a massively distributed way (e.g. hundreds of machines).

The method we designed for the distributed and approximate verification

of probabilistic systems is completely different since it is naturally a parallel method (due to the use of a Monte-Carlo sampling technique). There already exists other sampling techniques for the verification of probabilistic systems [16,4]. The method of [16] uses the framework of hypothesis testing while [4] uses also a Monte-Carlo method. These two methods have also the potential of being parallelized, but, to our knowledge, it wasn't done by now.

3 Approximate Probabilistic Model Checking

3.1 Theoretical Foundations

The APMC approach [5] uses an efficient Monte-Carlo method to approximate satisfaction probabilities of monotone properties over fully probabilistic transitions systems. Properties to be checked are expressed in Linear Temporal Logic (LTL).

3.1.1 APMC method

LTL formulas are built over a set of atomic propositions labeling states.

Definition 3.1 A fully probabilistic transition system (PTS or DTMC for Discrete Time Markov Chain) is a tuple $\mathcal{M} = (S, \overline{s}, P)$ where S is a set of states, $\overline{s}$ is the initial state, and P is a transition probability function i.e. for all $s \in S$, $\sum_{t \in S} P(s,t) = 1$.

We denote by $Path(s)$ the set of paths whose first state is s. The length of a path π is the number of states in the path and is denoted by $|\pi|$, this length can be infinite. The probability measure $Prob$ over the set $Path(s)$ is defined in a classical way [8]. We denote by $Prob[\phi]$ the measure of the set of paths $\{\pi \mid \pi(0) = s \text{ and } \mathcal{M}, \pi \models \phi\}$ (see [15]). Let $Path_k(s)$ be the set of all paths of length $k > 0$ starting at s in a PTS. The probability of an LTL formula ϕ on $Path_k(s)$ is the measure of paths satisfying ϕ in $Path_k(s)$ and is denoted by $Prob_k[\phi]$.

Definition 3.2 An LTL formula ϕ is *monotone* if and only if for all $k > 0$, for all paths π of length k, $\mathcal{M}, \pi \models \phi \implies \mathcal{M}, \pi^+ \models \phi$, where π^+ is any path of which π is a prefix.

A basic property of monotone formulas is the following one: if ϕ is a monotone formula, $0 < b \leq 1$ and if there exists some $k \in \mathbb{N}^*$ such that $Prob_k[\phi] \geq b$, then $Prob[\phi] \geq b$.

In order to verify some probabilistic specification $Prob[\phi] \geq b$, we choose a first value of $k = O(log|S|)$, then we approximate the probability $Prob_k[\phi]$ and test if the result is greater than b. If $Prob_k[\phi] \geq b$ is true, then the monotonicity of the property guarantees that $Prob[\phi] \geq b$ is true. Otherwise, we increment the value of k and approximate again $Prob_k[\phi]$. We iterate this procedure within a certain bound which, in many cases, is logarithmic in the

number of states. In the worst case, this bound is strongly related to the rapid mixing rate of the underlying Markov chain [12]. If the results of all tests $Prob_k[\psi] \geq b$ are negative, then we can conclude that $Prob[\psi] \not\geq b$. If we are interested only with probabilistic time bounded properties, as here, we can set k to the maximum time bound in subformulas of the specification. In the following, we describe how to approximate efficiently the probability $Prob_k[\phi]$.

3.1.2 Randomized approximation scheme

In order to estimate the probabilities of monotone properties with a simple randomized algorithm, we generate random paths in the probabilistic space underlying the DTMC structure of depth k and compute a random variable A/N which estimates $Prob_k[\psi]$. To verify a statement $Prob_k[\psi] \geq b$, we test whether $A/N > b - \varepsilon$. Our decision is correct with confidence $(1-\delta)$ after a number of samples polynomial in $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. The main advantage is that, in order to design a path generator, we need only a succinct representation of the transition graph, that is a succinct description in an input language, which is the same as in PRISM (Reactive Modules [1]). Our approximation problem is defined by giving as input x a succinct representation of a DTMC, a formula and a positive integer k. The succinct representation is used to generate a set of execution paths of length k. A randomized approximation scheme is a randomized algorithm which computes with high confidence a good approximation of the probability measure $\mu(x)$ of the set of execution paths satisfying the formula ϕ.

Definition 3.3 A fully polynomial randomized approximation scheme (FPRAS) for a probability problem is a randomized algorithm $\mathcal{A}$ that takes an input x, two real numbers $0 < \varepsilon, \delta < 1$ and produces a value $A(x, \varepsilon, \delta)$ such that:

$$Prob\big[|A(x,\varepsilon,\delta) - \mu(x)| \leq \varepsilon\big] \;\geq\; 1-\delta.$$

The running time of $\mathcal{A}$ is polynomial in $|x|$, $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$.

The probability is taken over the random choices of the algorithm. We call ε the *approximation parameter* and δ the *confidence parameter*. The APMC approximation algorithm consists in generating $O(\frac{1}{\varepsilon^2}.\log\frac{1}{\delta})$ paths, verifying the formula ϕ on each path and computing the fraction of satisfying paths, that is an ε-approximation of $Prob_k[\phi]$.

Theorem 3.4 *The APMC approximation algorithm is a fully randomized approximation scheme for the probability $p = Prob_k[\phi]$ of an LTL formula ϕ if $p \in]0,1[$.*

This result is obtained by using Chernoff-Hoeffding bounds [6] on the tail of the distribution of a sum of independent random variables. The time complexity of the algorithm is polynomial in $\log(1/\delta)$ and $1/\varepsilon$. The space complexity is linear in the length of execution paths.

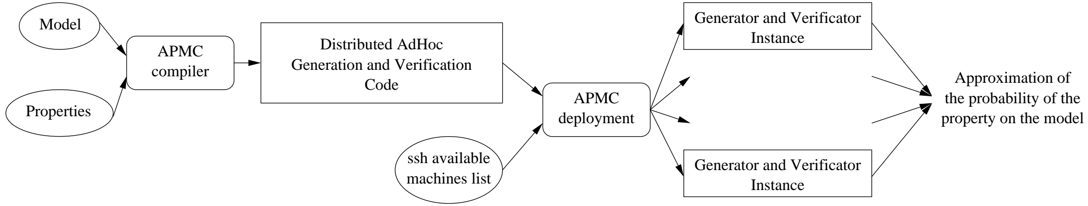


Fig. 1. APMC components

3.2 Architecture of APMC

APMC architecture is twofold, as described in figure 1. The first component, the APMC Compiler, produces an ad-hoc verifier including a sample generator and a checker for a given model (described in Reactive Modules) and a given property (LTL). The second module, the APMC Deployer, takes this verifier and the set of available computing resources, deploys the verifier on this set of computers and collects the result, which is the approximated value of satisfaction probability of the formula on the model.

The technique used to approximate this value assumes the verification of the formula on a large set of independent samples of bounded length. We use the independence property of the samples to distribute the generation and verification of samples.

The deployment is performed following a spanning tree of bounded arity. Each node of the tree runs on a single computing resource, and spawns children up to the bound on other available resources. While its parent still accepts results from it, and until the number of collected samples is greater than the requested number if it is the root, it generates a sample and verifies the property on it. At each verification, the counters of false and true samples are updated. Regularly (that is on a fixed timeout), each node sends its counters of false and true samples to its parent, and resets them (except for the root, which awaits the end of the computation to produce these numbers). When a node receives these counters from one of its children, it aggregates these numbers as if it produced the verification (see figure 2).

This deployment technique is assumed to be scalable, since the number and amount of data of all communications concerning a given node depends only on the arity bound of the tree. The tree topology was chosen to reduce the starting time, which is proportional to the depth of the tree, hence logarithmic in the number of computing resources. It also provides a logarithmic latency to aggregate the results from all nodes in the root. A drawback of this method is that the system may over generate and verify some samples (which does not preclude the validity of the final result, but may provide a better approximation than requested), up until the root claims that enough samples have been generated, and the tree is destroyed. This diffusion is also linear in the height of the tree and proportional to the communication timeout.

As for the parallelization, the technique provides a simple solution for fault tolerance: since each generation and verification is independent from

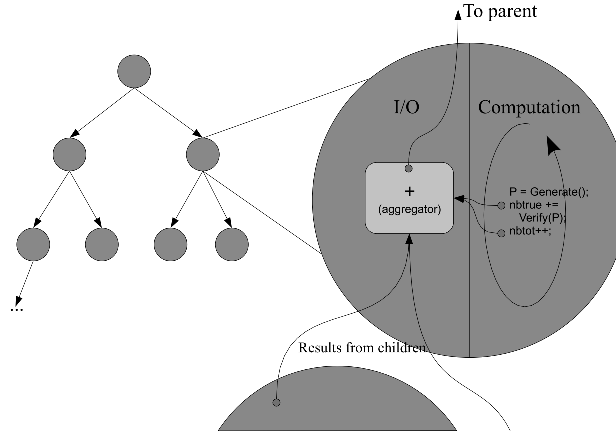


Fig. 2. APMC deployment scheme

the others, some of these verifications may be lost without consequences on the quality of the result. Thus, if a computing node crashes, its children will presume that the computation is finished and will stop running; its parent detects it and simply spawns a new subtree. All the workers of the subtree rooted at the crashed process are assumed lost and free to use again.

3.3 Implementation

The APMC software consists of three independent components: the parser, the core library and the deployment tool. This design provides the possibility to include the engine (core library) in many model checkers, like we are doing with the integration into the PRISM tool [11].

The parser is a simple lex/yacc program which parses a sub-language of the PRISM language (Reactive Modules [1]), and a simple language for LTL formulas. It then calls the core APMC library to produce an internal succinct representation of the model (linear in the size of the Reactive Modules file), and of the properties (linear in the size of the property file).

The library then produces the ad-hoc generator and verifier as ANSI C code (the generator/verifier is a standalone program deployed by the Deployer). APMC implements three strategies to generate the code of this program with respect to the synchronizations of the Reactive Modules: the first one (called *sync at compile-time*) pre-computes all the combinations of rules, thus building the synchronized succinct model representation, where each rule is not synchronized. This is the most efficient strategy with respect to time, but it is the most memory consuming strategy. At runtime, the generator simply evaluates each guard on the current configuration, building the set of fireable rules. A rule is chosen randomly between these fireable rules and the action is triggered to compute the next configuration. The second strategy (*sync at run-time*) is provided to handle larger, highly synchronized, models. There, the evaluation of the guards is done together with the computation of synchro-

nizations, which is thus done at each simulation step, spending more time to compute the set of fireable rules, but using much less memory. This strategy is used only when the model induces a lot of synchronization and the generated code prohibits efficient compilation. The last strategy is an improvement of the first one: for some models, at each step, most of the rules are fireable. When this is the case, instead of first computing the fireable rules (thus evaluating each guard on the configuration), a rule is chosen uniformly, and if its guard is true, its action is triggered. If its guard is false, another rule is chosen randomly.

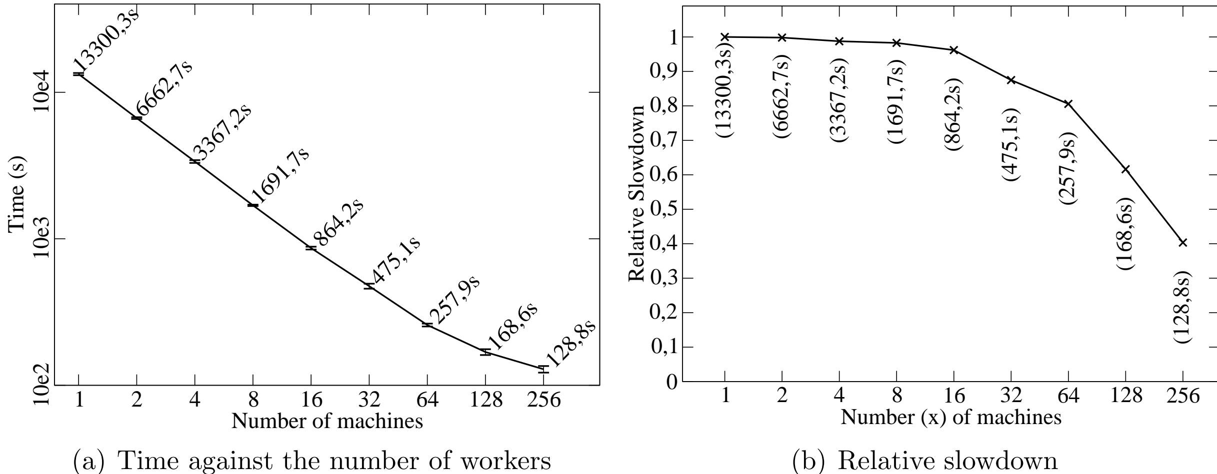
The main loop of the code produced by the library consists of generating a path (i.e. a set of configurations) of given length, and evaluating the property (linear time formula) on each path. The number of iterations of this loop is a parameter to the program.

The last component of the APMC software is the deployment tool. This tool takes the code produced by the library, compiles it on different architectures and deploys the programs on a set of computing nodes following a regular spanning tree of bounded arity. The program executed on the nodes includes two parts: an I/O part, and a computing part. The computing part is generated by the core library, while the I/O part is generic. This I/O part implements the spanning tree. It handles the connections with the children and with the parent of the node. Parent connection is handled through the standard output. Messages are sent regularly to the parent, according to the algorithm described in the architecture section. When this file descriptor is closed, the computation is stopped and the program exits. Children connections are handled using a double pipe with an ssh (or rsh) command. The deployment tool comes with a set of shell scripts passed to the ssh command. These scripts download and compile for the new spawned computing node the generated code, split the list of available resources between the children and launch recursively the compiled program on the node. This technique does not presume the existence of NFS, or other file sharing system. Currently, we assume that each node provides a remote shell service (ssh or rsh), and the autotools, Make and a C compiler. Current work in progress will assume only the C compiler and will reduce the amount of needed compilations by factorizing the compilations for each kind of architecture, instead of doing a compilation on each machine.

4 Performance Evaluation

The experimental platform consists of 500 Athlon 3000+ workstations with 1Gb of RAM, running under NetBSD 1.6.1, 100Mb ethernet network. The remote shell program used is OpenSSH, with public key authentication. The compiler on each worker is gcc-2.95.3, with the -03 option.

All the measurements are done on the dining philosopher problem, checking a double accessibility property. The dining philosopher problem [14], being



(a) Time against the number of workers

(b) Relative slowdown

Fig. 3. Time of model checking for the randomized dining philosophers problem

well studied, allows us to separate between the phenomenons due to the tool and those due to the model itself. Since this model does not include synchronizations, we conducted all experiments using the most efficient strategy, "sync at compile-time".

The first set of figures (figure 3) describes the acceleration in time due to the parallelization. To obtain these results, we ran APMC on the 160 dining philosopher problem [14], on an increasing number of workers following the binary tree deployment described in section 3.2. For all these experiments, we set $\epsilon = 10^{-2}, \delta = 10^{-10}$ (that is a generation of 940,000 paths by experiment), which are common values for these parameters, and $k = 200$. On the curves are represented the mean value of a set of 80 measures by point.

The first figure 3(a) shows the total execution time as function of the number of workers on a double logarithmic scale. One can see that, as expected, the execution time decreases quickly as the number of workers increases. The figure also shows a slowdown in the linear acceleration when having more than 64 workers.

The next figure 3(b) focuses on this phenomenon. The x axis represents the number of workers, and the y axis the relative slowdown given by the formula $y_x = t_1/(x \times t_x)$ where t_x is the time measured in figure 3(a) for the given x. With this measure, the value 1.0 represents perfect scalability, whereas smaller values demonstrate a lower use of the whole system.

One can see that when using more than 32 workers, the relative slowdown is higher than 10% on this example. The deployment phase is time consuming, and starting at 32 workers, the deployment duration is not negligible compared to the computation time. This accumulated time consumption is exponential in the depth of the tree (that is linear in the number of workers), nonetheless each worker waits at most for a logarithmic time before beginning its execution, which explains why adding workers is an improvement up to the amount where the computation ends before launching the last workers.

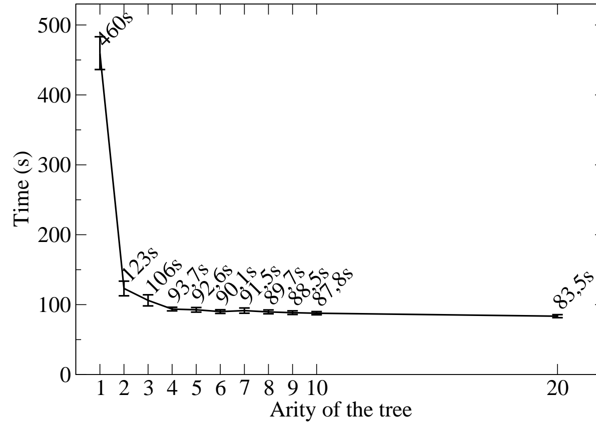


Fig. 4. Time of model checking according to the arity of the deployment tree

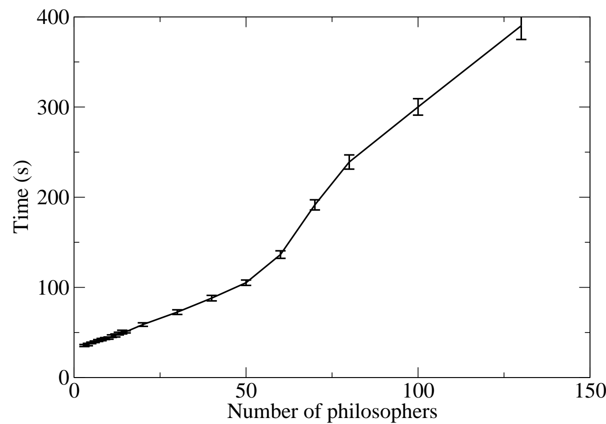


Fig. 5. Time of model checking against the number of philosophers

Figure 4 shows the time needed to verify the model with the same parameters as in figure 3 on a cluster of 256 workers, as function of the arity of the deployment tree. Obviously, except the case of arity 1 (a string of workers), increasing the arity of the tree does not improve significantly the performances of the deployment system. On the other hand, increasing the arity does not hinder the performances, so as the figure 3 teaches us, when the relative slowdown becomes too large, it makes sense to increase the arity in order to decrease the depth of the tree.

The last figure 5 presents the time needed to verify the 3 to 130 dining philosopher problems. All verifications were done on paths of length 200. 32 workers were used to verify 940,000 paths. The aim of this experiment is to evaluate the generation of the code. Indeed, since all verifications use the same path length and the same number of paths, the time differences are only due to the quality of the generated code.

One can see that the curve is in three linear parts. The main loop of the code consists in iterating over all the guards of the model (which are functions

of the program). The number of guards increases linearly with the number of philosophers. So it is natural that the time needed to iterate over all the guards is linear in the number of philosophers. It is less expected that the curve presents three different slopes. Since the generated code occupies up to 256Kb more resident memory for the 130 philosophers problem than for the 3 philosopher problem, we suspect that this is due to CPU code cache invalidations.

As a validation of APMC as a cycle stealing verification tool, we conducted another experiment including all the available computers of the EPITA school of computer science. We verified the 160 dining philosophers problem on a platform of 500 computers used by other applications at the same time. We conducted two experiments: the first where 940,000 paths were generated, the other one with 9,400,000. The first experiment took 99 seconds, the second one 446 seconds.

It is interesting to note that, although the amount of computation needed in the second experiment was ten times higher than for the first, the time needed to complete it was only 4.5 times higher. It is due to the fact that for the first experiment, the system does not have enough time to take advantage of the full platform.

5 Discussion

Traditionally, model checking is a highly expensive computational activity. The main drawback of the method is the memory needed to finalize the verification of large systems. "Classical" distributed model checking aims to lower the memory cost by distributing the state space. Using approximation techniques, we can trade the memory cost with simple computations on a large number of system executions paths. This is the point where we can massively distribute the process, by partitioning the sample into sets that are independently processed.

Using this method, we can verify very large systems using a constant amount of memory (when the length k is fixed). The power of computation usable for the verification is limited only by the number of available computers.

However, experiments show that for each system, there is a critical number of machines after which the time needed for the verification may not decrease significantly. This is due to the non negligible cost of the deployment scheme, which is a function of the depth of the tree. Other experiments showed that increasing the arity of the deployment tree may reduce the depth with small performance cost. There is a trade-off between the depth of the tree and its arity. With a high depth, there will not be any communication bottleneck for the nodes of the tree, while with a high arity, the communication load on each node will be higher. Nonetheless, since the amount of communication is low, one can choose a reasonable arity without losses of performances.

APMC is also interesting from an economic point of view. Since APMC runs in background using few memory, it can run on classical desktop machines (implementing cycle stealing techniques), thus avoiding the cost of an expensive cluster of dedicated workstations.

References

[1] R. Alur and T. Henzinger. Reactive modules. in Proc. of *the 11th Annual IEEE Symposium on Logic In Computer Science (LICS)*, IEEE Computer Society Press, pp 207-218. 1996.

[2] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer-Aided Design, Third International Conference, (FMCAD'00)*, volume 1954 of LNCS, pp 390–404, 2000.

[3] H. Garavel, R. Mateescu and I. Smarandache. Parallel state space construction for model-checking. Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada), volume 2057 of LNCS, pages 217–234, 2001.

[4] R. Grosu and S. A. Smolka. Monte Carlo Model Checking. Proceedings of *11th Tool and Algorithms for the Construction and Analysis of Systems (TACAS 2005).* pp 271–286, LNCS 3440. 2005.

[5] T. Herault, R. Lassaigne, F. Magniette and S. Peyronnet. Approximate Probabilistic Model Checking. *Proceedings of Fifth International VMCAI'04*, pp 73-84, LNCS 2937, January 2004.

[6] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13-30, 1963.

[7] C. P. Inggs and H. Barringer. On the parallelisation of model checking. *Proceedings of the 2nd Workshop on Automated Verification of Critical Systems (AVOCS'02).* Technical report, University of Birmingham, April 2002.

[8] J. Kemeny, J. Snell and A. Knapp. *Denumerable markov chains.* Springer-Verlag, 1976.

[9] W. J. Knottenbelt and P. G. Harrison. Distributed disk-based solution techniques for large Markov models. Proc. of NSMC'99, 3rd International Workshop on the Numerical Solution of Markov Chains, September 1999.

[10] M. Kwiatkowska, R. Mehmood, G. Norman and D. Parker. Symbolic Out-of-Core Solution Method for Markov Model. Proc. Workshop on Parallel and Distributed Model Checking (PDMC'02), volume 68.4 of ENTCS, August 2002

[11] M. Kwiatkowska, G. Norman and D. Parker. PRISM: Probabilistic Symbolic Model Checker. *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, pp 200–204, LNCS 2324, 2002.

[12] L. Lovasz and P. Winkler. Exact mixing time in an unknown markov chain. *Electronic journal of combinatorics*, 1995.

[13] D. M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *J. Parallel Distrib. Comput.*, 47(2):153-167, 1997.

[14] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, pages 1:53–72, 1986.

[15] M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. *Proc. 26th Annual Symposium on Foundations of Computer Science*,pp 327–338. 1985.

[16] H. L. S. Younes and R. G. Simmons. Probabilistic Verication of Discrete Event Systems using Acceptance Sampling. *Proc. of the 14th International Conference on Computer Aided Verification*, LNCS, 2404:223–235. 2002.

Under-approximation heuristics for Grid-based BMC

Subramanian Iyer

Department of Computer Sciences, University of Texas at Austin
Austin, Texas, USA

Jawahar Jain

Fujitsu Laboratories of America, Inc.
Sunnyvale, California, USA

Debashis Sahoo

Department of Electrical Engineering, Stanford University
Stanford, California, USA

E. Allen Emerson [1]

Department of Computer Sciences, University of Texas at Austin
Austin, Texas, USA

Abstract

In this paper, we consider the effect of BDD-based under-approximation on a hybrid approach using BDDs and SAT-BMC for error detection on a computing grid. We experimentally study effect of under-approximation approaches on a non-traditional parallelization of BMC based on state space partitioning. This parallelization is accomplished by executing multiple instances of BMC independently from different seed states, that are selected from the reachable states in different partitions. Such states are spread out across the state space and can potentially be *deep*. Since all processors work independently of each other, this scheme is suitable for bug hunting using a grid-like network. Our experimental results demonstrate improvement over existing approaches, and we show that the method can effectively utilize a large grid network.

Key words: BDD, SAT, Bounded Model Checking, Parallel Computing, Grid Computing

[1] Prof. Emerson thanks the NSF for support via grants CCR-009-8141 and CCR-020-5483.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Formal verification, especially error detection, is rapidly increasing in importance with the rising complexity of designs. The main constraint in verification is the total amount of resources available, time as well as memory.

Most attempts at verification only use a single processor. Recently, various attempts have been made to use parallel and distributed methods for verification. All these approaches assume the presence of a dedicated network of workstations to perform verification tasks.

As "personal computers" gain in computing capacity, the concept of computation grids is gaining acceptance [11]. Here, a *grid* is a network of machines that are not dedicated to a specific computational use, but may only be available some of the time. This is a unique environment where massive parallelism is possible by using otherwise idle CPU cycles from a large number of computers. Such processors may even be in geographically diverse locations.

The issues in a grid-computing environment are quite different from those in dedicated parallel computing environments. We consider two key issues. Firstly, the availability of the processors is not guaranteed. So any algorithm that uses such a framework has to be able to withstand receiving either no results or only partial results from certain computations. Secondly, since the computational network is not dedicated at all times to a single task, any task scheduled on a grid has to use very little network bandwidth. As far as possible, computations on different nodes need to be independent of each other, with very few dependencies.

Under such circumstances, algorithms need to be carefully devised in order to scale to grid-based parallel networks. Our work specifically targets the issue of how to effectively use a grid for formal verification. For failing properties that are not very deep, Bounded Model Checking (*BMC*) is now the *de facto* standard. For proving the property correct or for finding deep bugs, a different method, typically a BDD based approach, is generally the choice, and works well if image computation can be performed efficiently. Since one typically does not know *a priori* whether a property is erroneous or not, and if erroneous, whether it is deep, both methods have to be run on every property. We detail a practical and efficient grid based approach to error detection that is designed to automatically handle deep as well as shallow bugs.

Current Approaches to Error Detection

Satisfiability based Bounded Model Checking (SAT-BMC) is able to explore the state space of larger designs by bounding the depth of exploration and successively increasing this bound [8]. Due to notable improvements in the art of satisfiability-testing, SAT-BMC is now routinely applied to detect errors during property verification for many industrial designs [9,4,3,1].

SAT based BMC approaches are the preferred method for detecting error states that are not very deep. However, these techniques can become quite expensive when many time-frames are required to be analyzed. BDD based approaches work better for those "deep cases" as long as the image BDDs remain moderately small. These

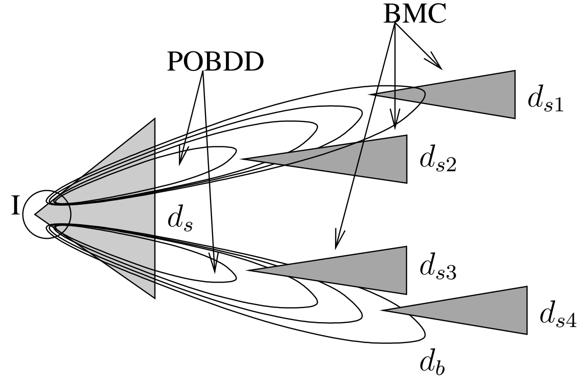


Fig. 1. Seeding multiple SAT-BMC runs from POBDD reachability

observations have been validated in a recently reported industrial case study [1]. Thus the class of problems that require many steps of image analysis to detect the error, *but* where BDD sizes grow large, remains an attractive research target.

Since such techniques, running on a single CPU have clear limitations due to the limited computing power of their execution environment, researchers have suggested distributed approach towards BDD based model checking as well as parallel SAT solvers [10,13]. However these methods remain inadequate as they essentially analyze the state space from a breadth-first search point of view. The BDD based approaches also require communication between different processors in form of large BDDs which precludes these methods from using large parallel computing environments.

Our approach: Grid-BMC

Our approach to locating errors is to create a method that finds various candidate deep reachable states which are then used as *seeds* for running many instances of SAT-BMC in *parallel* to explore the state space adjacent to such seeds. Starting from such potentially deep seed states, multiple BMC runs may be able to reach further deep states and locate errors that are otherwise not locatable by existing methods. Partitioned BDDs (POBDDs) [21] have a great potential for compactness, as studied in the literature [6,24]. At the same time, as we shall see later, they are very sensitive to even minor changes in parameter settings. From our empirical observations, we suggest an approach that is able to systematically examine multiple POBDD representations constructed and analyzed independently on different nodes of a grid. This leads to multiple traversals of states in orders, often significantly differing from the standard BFS exploration and is found to greatly benefit the process of error detection.

Figure 1 shows this pictorially using two partitions and four instances of SAT. The triangles represent a search using SAT, and the ellipses denote successive image computations using BDDs. Notice that from the initial states, BMC can only proceed to depth d_s effectively. Instead, in the proposed approach, BDDs go to a depth d_b, and many instances of SAT are seeded until then, which may be effective to differing depths $d_{s1}, d_{s2} \ldots d_{sn}$. Consequently, this can reach errors that are otherwise difficult to catch.

In this paper, we study the effect of varying the coarseness of this under-approximation used to locate states using BDD-based reachability.

2 Related Work

This work lies at the cross-roads of two bodies of work, namely hybrid techniques for smart simulation or efficient bug-finding and recent early efforts for performing verification in a parallel framework. The techniques discussed in this paper have a nature of an hybrid approach using multiple engines. Thus, they can easily use improvements in the individual technologies such as SAT or ATPG engines [23,20], or BMC formulation [8]. Thus, a detailed comparison with such techniques is orthogonal to the objective of our paper and is not further detailed.

Grid-BMC in the context of other hybrid or parallel approaches

This work differs from other hybrid approaches in two key aspects. Firstly, simulation forms the search backbone of many of the above methods. In the case of hard, deep bugs, a simulation based approach may not be able to access interesting regions of the search space. Secondly, our hybrid approach is formulated with the specific aim of being able to generate multiple state traversals that, when considered together, can potentially cover the entire state space, and may be performed independently in parallel.

In [29] a pre-image computation from the target states is used to provide an enlarged target for simulation. The **SIVA** tool [12] performs best first search in a simulation environment, augmented with BDDs and SAT, with the hamming distance between the current and target state as the guiding cost function. The approach of [19] uses interleaved runs of simulation, test generation, BDD-based symbolic simulation and SAT-based BMC to maximize the state coverage over a set of interesting signals.

In contrast to the above "bug hunting" approaches another class of techniques employ a combination of formal engines mainly for the purpose of verifying properties (possibly bounded depth properties). For example, [7,15] use BDD-based reachability analysis and [16] compute CNF clauses through BDD functional analysis to prune the search space of SAT-based BMC, [22] structurally partition the property check into parts solved by SAT and BDDs and [17] employ a combination of symbolic trajectory evaluation and SAT/BDD based model checking.

The previous discussion is in the context of a single processor framework. Typical hybrid approaches do not naturally lend themselves to parallel exploration. Our approach executes, in parallel, multiple independent BMC instances to concurrently explore different regions of the state space. This is a non-traditional parallelization of SAT-BMC.

Several other methods have been proposed to do parallel verification. Stern and Dill [26] parallelize an explicit model checker. In [27], parallelized BDDs are used for reachability analysis. Verification using parallel reachability analysis has been studied in [14,18,28]. Our work is different from other distributed model checking approaches which are geared towards completeness rather than bug hunting. Most techniques such

as [18] are to a large extent only parallelizing the breadth-first traversal. Thus their limitations to reach deep states remains a severe handicap. Further, these techniques require message passing between different processors in form of large BDDs. That can severely limit how large a grid can be effectively employed. A method proposed in [13] distribute the SAT-based BMC over a network of heterogeneous workstations. Their algorithm performs distributed BCP to solve a large SAT problem. Similarly, in [10] a parallel multi-threaded SAT solver is discussed.

3 The Grid Framework

We used the grid middle-ware CyberGrip [2], developed at Fujitsu Labs Limited, Japan, to manage the computing resources on grid. Figure 2 shows the overall ar-

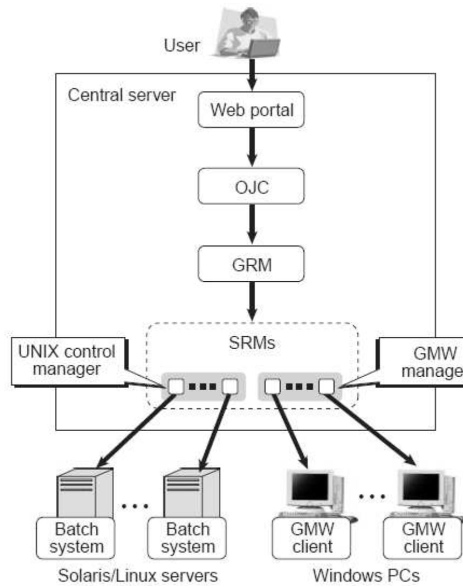


Fig. 2. Architecture of CyberGrip

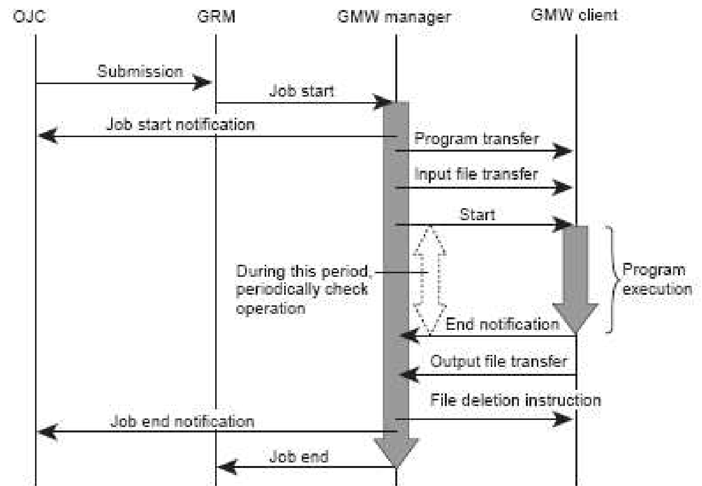


Fig. 3. Execution of a job on the Grid

chitecture of CyberGrip. CyberGrip operates on a central UNIX or Linux server and consists of three components: Organic Job Controller (OJC), Grid Resource Manager (GRM), and Site Resource Manager (SRM). The interaction between various components of CyberGrip in executing a job on the grid is depicted in Figure 3.

OJC, shown in Fig 4, controls how the jobs submitted by the user are executed. GRM determines the optimum computing resources for the jobs transferred from OJC. SRM monitors the status of the computing resources and manages the communication between them. There is one SRM for each computing resource, and the SRM for a Windows PC is called the Grid Mediator for Windows (GMW) manager. Each computing resource must have middle-ware to communicate with its SRM and execute jobs. When the computing resource is a Solaris or Linux machine, a general batch system, for example, Condor, can be used as the middle-ware.

CyberGrip can realize an environment in which the user can submit jobs to virtualized computing resources consisting of not only Solaris and Linux machines but

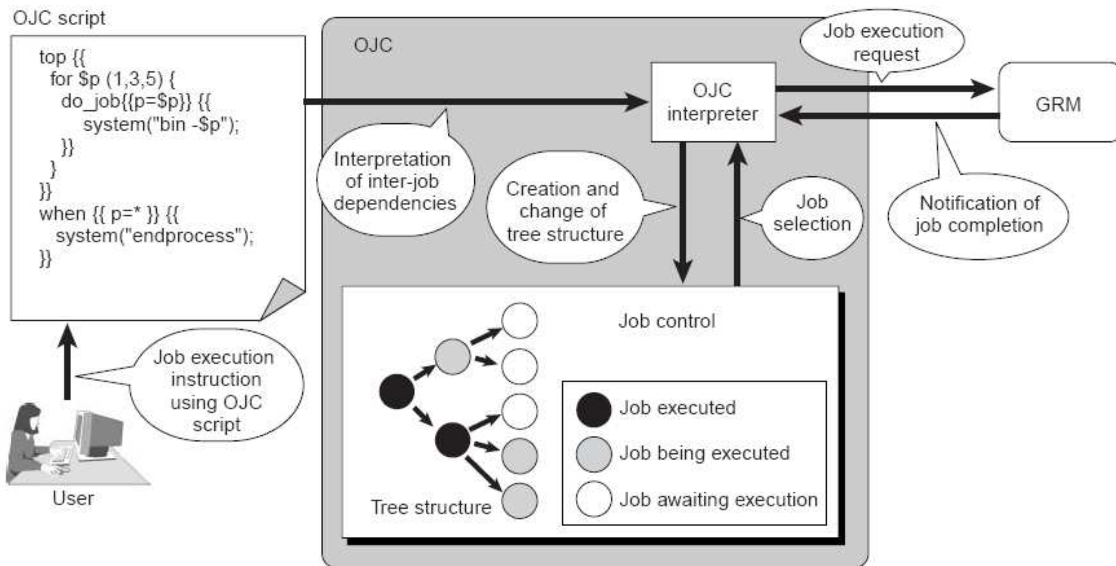


Fig. 4. Basic structure of Organic Job Controller (OJC)

also Windows machines for office use via the Web portal of a central server. The user can do this without being aware of the performance and other characteristics of the individual computers.

OJC also has a dynamic job control function shown in Fig 5. This function works

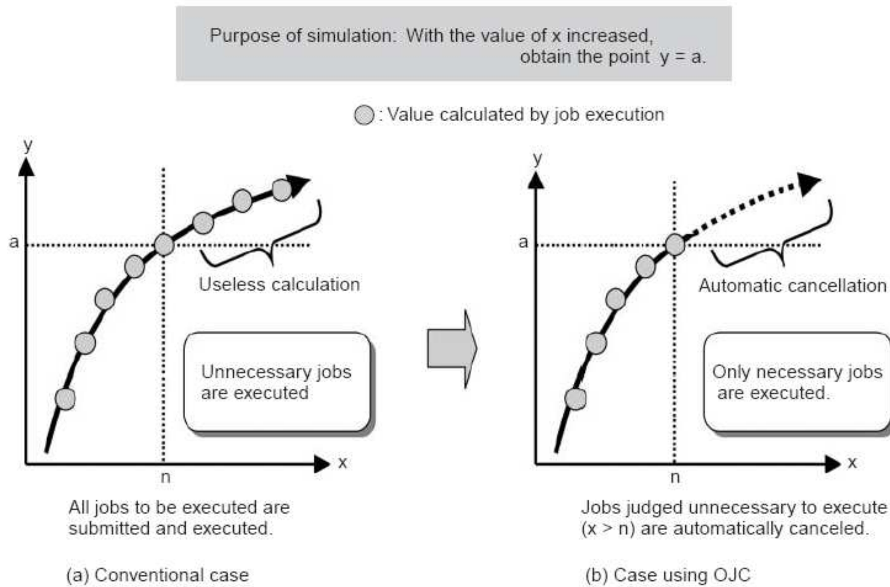


Fig. 5. Dynamic Job Control

when the number of jobs to be executed cannot be decided statically at initial job entry and the number of jobs to be executed varies dynamically. Conventionally, because it was difficult to automatically perform dynamic job control, operators usually took

one of two approaches: 1. execute all the jobs that have been submitted without considering which ones need to be executed or 2. individually decide whether to execute each job based on the execution results of the previously executed job. Both these approaches are very inefficient. By automating these decisions, therefore, OJC significantly increases the efficiency of job execution. This is critical for our usage of the grid – once an error is detected, we want all nodes on the grid to cease working on the problem.

4 Under-approximation based Grid-BMC

As discussed earlier, our paper specifically targets the problems where deep-states have to be explored and current BDDs and BMC methods may be inadequate. BMC based on SAT has a limitation on how deep a state it can explore as it is based on explicitly unrolling multiple time frames, equal in number to the length of the suspected error path. BDDs calculate successive image computations for reachability and can go deep, provided, the size of the transition relation is manageable and the successive images are small in size. Unfortunately, such images often get large, at a very small depth, and BDDs are unable to make further progress. Even if BDDs do not exhibit dramatic blowup in size, the computed image sets grow in size steadily, until they are so large that the calculations need impractically long time. Thus, for smaller depths SAT may be able to proceed further as it does not store sets of states, instead merely computes paths. We suggest a method for exploring deep states in the following.

4.1 Key Idea: Under-approximation for Grid-BMC

In this section, we describe the under-approximation heuristics that are used to perform deep state space traversal. Under-approximation is performed at two different places – firstly, during reachability analysis and secondly, while selecting initial states for SAT-based BMC.

Find Deep States: We perform a traversal of the state space by partitioning the transition relation, as well as the computed sets of states so that the BDDs and associated calculations remain tractable. When the BDD sizes are no longer manageable, we perform successive under-approximations. At each step of image computation, we use a subset of the actual set of states. Such massive under-approximation may result in successive traversal not always leading to a deeper state. The quality of this approximation can be further improved, esp. with input from designer, for instance by using guided traversal [25,5]. Under-approximation allows some control over the size of BDDs, which can otherwise exhibit dramatic blow-ups. We find that the above simple approach is quite effective and study it in detail in this paper.

Parallel Seed SAT: In order to determine the initial seed states for SAT, a large number of partitions are explored very rapidly with under-approximation, and the resulting deep states are written out at regular intervals, as CNF clauses. Currently, we create a new seed after a fixed number image computations, say, after every 5 images. In order to do this, a snapshot of the reachable states is taken and a subset

of those states is used to seed the SAT solver. It is critical to pick a small number of states and not all states, otherwise the SAT solver can choke as it gets a very large number of clauses. The SAT solver instance executes a BMC-like algorithm from the seed thus obtained. By making multiple BMC runs, starting from various points along the state traversal, we can ensure that at least a subset of the BMC executions start from a deep state. These may then explore regions that could not be explored using traditional SAT-BMC approaches. Since all BMC runs can be made in parallel so this leads to a non-traditional method of parallelizing BMC.

Typically we execute SAT-BMC to a small depth that can be computed in a matter of minutes using zchaff SAT solver. Note that it is critical to run these SAT instances separately rather than run a single instance of SAT from their union. The reason is that seed states from different portions of the state space may be very dissimilar and if they are combined together in generating the clauses for SAT, the effectiveness of the SAT solver may reduce drastically.

Degree of Approximation: Each partition can be considered as an automatic selection of "direction" with respect to exploration of states. Reachability with partitioning localizes the Breadth First Search along such directions. The under-approximation in this exploration is performed simply by picking a few random states from the set of new states found during each image computation. The size of BDDs generated is reduced by this using under-approximation albeit at the cost of loss of information. Keeping BDD sizes small in this manner allows for deeper exploration in selected "directions" and can be used to provide many different initial states to seed subsequent BMC runs. If any particular local BFS traversal leads it in a direction which corresponds to a bug in the design then a BMC started from corresponding seed can prove to be far more efficient then classical BMC started from the original initial state. Notice that there is a trade-off between going deep versus exploring all directions.

4.2 Approximation and Usage of Grid

There is a clear tradeoff between the degree of approximation and number of CPUs required. If many states are selected as a seed state, then effectively it corresponds to simultaneously searching in many directions with BMC. However, the corresponding BMC run may become slower. To solve this dilemma we propose the following method. The grid-BMC starts by dividing the grid into multiple sub-grids. One sub-grid uses severe under approximation to go very deep. The others run POBDDs and BMC with varying degrees of approximation. Then we monitor for either of the following condition to arise: (a) Though a more complex seed is used, the BMC runtime is not adversely effected; (b) Though a less complex seed is used, the number of nodes in the grid are rapidly used up. In such cases, the algorithm automatically switches to using the larger number of min-terms as seed for BMC on the main sub-grid. Otherwise, we continue to use the less complex seeds with fewer min-terms. In order to switch between approximation levels, the runs with a higher approximation are canceled and the corresponding nodes on the grid are freed for the runs that use lower approximation (more number of min-terms used as seed for BMC as well as frontier

for POBDD image calculation).

We divide the grid amongst BMC runs with different under-approximations. The question is how to dynamically decide which sub-grid is showing sub-optimal behavior so that its given run may be cancelled and the corresponding CPUs freed up. The quality of a run can be indicated by the following three resources consumed. A disproportionately large value for either resource in any of the sub-grids should lead to the run being aborted, and the nodes in that sub-grid being freed up.

(a) POBDD time: If in one of the sub-grids the BDD size (resp. time) starts increasing significantly as compared to the BDD size (time) in other sub-grid, then it indicates a run that is being hampered by suboptimal variable order or quantification schedule and should be aborted. Hence the time for each step of image computation acts as a good filter for the sub-optimal runs.

(b) Ratio of BMC time/time-frame: At times it is seen that the BMC runs from some seeds start consuming a disproportionately large time and become impractically slow. This generally happens when the number of states used as the initial seed state become very large. A ratio of the average runtime/time-frame can be maintained, and when in one of the grid this number is significantly exceeded then the corresponding run on the sub-grid is aborted.

(c) Number of CPUs used - Under extreme conditions, this should be used as another measure for tagging the runs on a given sub-grid to be sub-optimal. For greater under-approximation (fewer states in seed), usage of a larger number of CPUs decreases its attractiveness. This is because when initial seed state set is smaller, each BMC run is starting from the end of traversal in a very narrow direction. Hence, the use of a large number of CPUs potentially indicates a large number of unsuccessful BMC runs, which in turn implies that the starting seed states are not a good choice and are perhaps not in the area corresponding to the error.

4.3 Outline of Grid-BMC Algorithm

Divide_Grid: Create multiple sub-grids to dynamically determine a good approximation threshold in order to detect whether an error exists. On each sub-grid, do the following:

(i) *Partition_reach*: Use state partitioning in reachability to get different and divergent paths exploring state space.

(ii) *Approx_Partition_reach*: Target deep space traversals – from each frontier, select an under-approximation of the newly reached states to do the next image computation.

(iii) *Generate_seed*: At regular intervals, whenever a threshold is crossed, store a *seed* – a few reachable states.

(iv) *Start_Seeded_SAT*: From each seed, pass it as the initial state to a new instance of the SAT solver.

(v) *Run_in_Parallel*: Run all instances of SAT-based BMC to a small depth, as nodes become available on the grid.

(vi) *Abort_Sub_Grid*: If the BDD image time and SAT-BMC time for each time-

Ckt	Num. latches	Error Depth	Run-times (sec) for BMC seeded from simulation									
			2k		4k		6k		8k		10k	
			Sim	BMC	Sim	BMC	Sim	BMC	Sim	BMC	Sim	BMC
b1	125	59	22	215	25	207	48	151	56	66	63	196
b2	70	85	18	117	21	105	32	96	45	110	55	118
b3	66	23	21	333	25	195	38	232	50	258	63	258
b4	66	59	27	2211	26	2747	60	2442	56	2239	63	2060
b5	170	36	653	1047	923	2067	1605	1561	1822	1735	2333	2493
b6	201	29	629	487	921	509	1258	396	1756	346	2423	313
b7	123	60	892	T/O	1305	T/O	2951	T/O	2694	T/O	3515	T/O
b8	169	23	105	T/O	122	T/O	193	T/O	361	T/O	476	T/O
b9	148	27	106	T/O	130	T/O	295	T/O	256	T/O	462	T/O

"T/O" is a timeout of 2 hrs

Table 1
Run-times for BMC seeded from simulation to various depths.

frame in this sub-grid are disproportionately larger than other sub-grids, then abort runs on this sub-grid.
Termination_condition: Allow BDD and SAT explorations to continue in parallel on all sub-grids until error is found or timeout is reached.

5 Results

We now present experimental results that demonstrate the efficacy of our method. The experiments are run on a grid of computers that included up to 100 independent Xeon CPUs (ranging from 1.5 GHz to 2.3 GHz) running linux. As explained earlier, we used an in-house grid middle-ware (CyberGrip) developed at Fujitsu Labs Limited, Japan, for submitting and controlling jobs executed on the grid. Our program is implemented on top of VIS-2.0 and used CUDD BDD package and zchaff as the SAT-solver. The POBDD algorithm is run on a single processor but the CNF files generated are transfered to different nodes on the grid where a BMC run is fired in parallel. We were unable to exactly measure the time taken in transferring the files but in our experience it is very small.

Benchmarks: We used 9 circuits and properties, $b1, \ldots b9$, that were obtained during verification of a variety of industrial circuits. Several of these properties are deep and pose some difficulty for SAT-BMC as well as for simulation based methods. Thus they form a good benchmark for judging the efficacy of our approach.

5.1 Details of Experiments

Simulation: First we ran experiments based on random simulation, using the VIS-2.0 package. Simulation was done twice, first to 5,000 and then to 100,000 steps, but it is unable to find a bug in any of the circuits in the benchmark. Then, we used simulation to find deep states and seed BMC from there. This is similar to the approach of [19],

Ckt	Num. latch	Error Depth	Time (sec)								Grid #CPU used
			BDD	POBDD	BMC	Sim	Sim+ BMC	Grid-BMC (pickOne)			
								Seed	SAT	Total	
b1	125	59	7	3.2	T/O	NB	167	7	176	**183**	8
b2	70	85	3.4	2	T/O	NB	115	97	26	**123**	40
b3	66	23	1.9	1.3	T/O	NB	268	1	1	**2**	2
b4	66	59	1.9	1.3	T/O	NB	3097	12	228	**240**	8
b5	170	36	T/O	T/O	T/O	NB	2758	27	36	**63**	9
b6	201	29	3148	2857	T/O	NB	1407	156	20	**176**	3
b7	123	60	258	976	T/O	NB	T/O	35	429	**464**	14
b8	169	23	T/O	T/O	T/O	NB	T/O	198	55	**253**	28
b9	148	27	T/O	T/O	T/O	NB	T/O	280	1580	**1860**	70

"T/O" is a timeout of 2 hrs, "NB" means no bug found.

Table 2
Comparison of the time taken in seconds by various approaches.

except that we use a different random seed for each simulation depth. For each circuit, we run simulation, in steps of 1,000 from 2,000 to 10,000. When the depth is reached, we pick the state reached at the end of the simulation and seed SAT from there. To limit the amount of data, the results of this are shown in table 1 for depths 2k, 4k, 6k, 8k and 10k. We found that simulation, even when it seeds SAT at periodic depths of every 1000 steps is unable to find any bug in any of the circuits in the benchmark.

Grid-BMC: Next, in table 2, we compare the following methods against each other: BDD-based reachability (VIS-2.0 and CUDD); POBDD; BMC (using SAT-solver zChaff); Random simulation to 5,000 steps; Simul to 5,000 steps + An application of SAT solver. The methods are compared with a run of Grid-BMC, that has a 10 minute initial phase for POBDD based seed generation, and 2 hours for SAT. In the case of Grid BMC, there are many seed states, so the table shows how long it took for POBDD based reachability to discover the "best" seed state and time for the SAT-solver to find the bug from there. The final column shows how many CPUs of the grid were actually used. We allow each method to run until a time out of 2 hours. The results for all the methods are shown in table 2. Grid-BMC uses severe under-approximation by picking only one state in each image and for each seed. Note that Grid-BMC is the only method that is able to find the error in b8 and b9.

5.2 Analysis

Our results show a strong evidence of a positive synergy between the two key ideas – how to find initial states, possibly deep, and how to independently process multiple seeds. Specifically, from table 2, we note that:

- Grid-BMC can often find errors significantly faster than BDDs or POBDDs alone.
- Grid-BMC finds errors on circuits where BMC runs out of time.
- BDD based seeding of SAT solvers works well, and is often more effective than seeding using random simulation, which is widely accepted as one of the best strategies

Ckt	pickTwo				pickThree				pickFive			
	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU
b1	18	395	413	16	185	551	736	4	47	302	349	14
b2	450	29	479	10	197	52	249	7	4	42	46	24
b3	1	3	4	2	1	5	6	3	7	2	9	8
b4	18	505	523	21	23	624	647	21	19	316	335	15
b5	252	83	335	9	241	78	319	5	26	80	106	24
b6	154	24	178	9	324	19	343	6	191	24	215	4
b7	333	623	956	23	43	489	532	3	95	716	811	28
b8	167	7	194	27	93	104	197	3	91	7	98	8
b9	88	745	833	21	342	234	576	87	270	218	488	89

Table 3

Effect on performance (time in seconds) by relaxing the severity of the approximation

Ckt	pickTen				pickTwenty				pickFifty			
	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU
b1	223	406	629	28	19	756	775	27	155	546	701	14
b2	408	72	480	44	84	103	187	20	65	945	1010	18
b3	1	3	4	2	1	4	5	3	11	4	15	3
b4	8	362	370	9	6	447	453	5	48	613	661	7
b5	25	69	94	6	91	91	182	14	127	183	310	20
b6	150	23	173	6	155	27	182	4	157	44	201	4
b7	43	711	754	8	296	798	1094	26	176	1222	1398	10
b8	82	84	166	16	114	38	152	5	124	30	154	4
b9	145	698	843	28	175	1213	1388	28	11	2977	2988	34

Table 4

Effect on performance (time in seconds) by drastically relaxing the severity of the approximation

for industrial designs.

- On every example, Grid-BMC is superior to BMC, random simulation and a combination of the two; either in finding an error faster, or by finding an error that is not otherwise found.
- On examples that are BDD-friendly, Grid-BMC performs better than the other BMC techniques.

For some circuits such as b1, b2, b3 and b4 the error can be detected in the POBDD phase itself. At the time the error was detected, no BMC run that had been been fired had yet completed. Thus grid-BMC is not required for these entries, but we show it so we can analyze the effect of approximation in further tables.

5.3 Effect of under-approximation

Table 3 shows the effect of relaxing the approximation, by picking more states at each step (call this m). In our experimental runs, we varied m from 1 to 5 for each circuit. Table 4 shows the effect of drastically relaxing the under-approximation, by

successively using 10, 20 and then 50 states at each step.

The above results can be viewed in the context of running two or more configurations (conceptually each can be conceived as a sub-grid) in parallel (say, $m = 1$ and $m = 5$). We suggest as approach to automatically decide the better configuration. Our decision approach needs to monitor for each sub-grid, the corresponding BDD image-time, and BMC-SAT time for each time-frame. If these runtimes become disproportionately and significantly larger then the given configuration is displaying sub-optimality. Note that as the under-approximation is relaxed, but the corresponding BDD image-time, and BMC-SAT time for each time-frame do not proportionately increase, then the more accurate approximation starts yielding faster error detection. This is logically expected since when we relax the severity of approximation, the resulting BMC can be deemed as searching simultaneously in multiple directions, and thus a larger state space. The utility of such an approach is confirmed for almost all cases by the experimental results presented here. For example, entry $b9$ requires 1860 seconds on the grid for $m = 1$. As m increases progressively from 1 to 5; the run-times proportionally decrease by a factor of 4. Identical observations hold for circuit $b8$ too.

6 Analysis and Conclusions

Grid-BMC for error detection is practical and effective. It is computationally inexpensive in terms of overhead and an alternate way of parallelizing SAT-based BMC – each of many processors can execute a BMC from a different set of initial states. The only data that is passed over the network is at the very beginning, after that no synchronization is required, until termination. Such parallelization has no interdependence at all, and can therefore very effectively utilize a number of processors in a large grid, without creating communication overhead between the processors. This method also effectively exploits the advantage of symbolic BDD based search as well as SAT as well as overcomes their respective limitations. For example, if there are a large number of partitions or if certain partitions are difficult, performing cross-over images between them can be difficult, and this may be the bottleneck in getting to the error. This can be overcome by SAT based BMC, which is "locally complete" from its originating point and does not compute sets of states. Although a very large grid ($\tilde{1}00$ nodes) was available, in typical experiments only a small number of CPUs were used. This suggests significant scope to improve the quality of results and possibility to tackle larger problems with further research.

References

[1] Amla, N., R. Kurshan, K. McMillan and R. Medel, *Experimental Analysis of Different Techniques for Bounded Model Checking*, in: *TACAS*, Lecture Notes in Computer Science **2619** (2003), pp. 34–48.

[2] Asato, A. and Y. Kadooka, *Grid Middleware for Effectively Utilizing Computing Resources: CyberGRIP*, in: *Fujitsu Scientific and Technical Journal*, 2004.

[3] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs*, in: *Proc. of Computer Aided Verification*, Lecture Notes in Computer Science **1633** (1999), pp. 60–71.

[4] Bjesse, P., T. Leonard and A. Mokkedem, *Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers*, in: *Proc. of Computer Aided Verification*, Lecture Notes in Computer Science **2102** (2001), pp. 454–464.

[5] Bloem, R., K. Ravi and F. Somenzi, *Symbolic guided search for CTL model checking*, in: *Proc. of the Design Automation Conf.*, 2000, pp. 29–34.

[6] Bollig, B. and I. Wegener, *Partitioned BDDs vs. other BDD models*, in: *Proc. of the Intl. Workshop on Logic Synthesis*, 1997.

[7] Cabodi, G., S. Nocco and S. Quer, *Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals*, in: *Proc. of the Design Automation and Test in Europe*, 2003, pp. 898–903.

[8] Clarke, E., A. Biere, R. Raimi and Y. Zhu, *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design **19** (2001), pp. 7–34, kluwer Academic Publishers.

[9] Copti, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Y. Vardi, *Benefits of Bounded Model Checking in an Industrial Setting*, in: *Proc. of Computer Aided Verification*, 2001, pp. 436–453.

[10] Feldman, Y., N. Dershowitz and Z. Hanna, *Parallel multithreaded satisfiability solver: Design and implementation*, in: *Workshop on Parallel and Distributed Methods in verifiCation*, 2004.

[11] Foster, I. and C. Kesselman, "The GRID: Blueprint for a new Computing Infrastructure." Morgan Kaufmann, 2003.

[12] Ganai, M., A. Aziz and A. Kuehlmann, *Enhancing Simulation with BDDs and ATPG*, in: *Proc. of the* 36th *Design Automation Conference*, 1999, pp. 385–390.

[13] Ganai, M. K., A. Gupta, Z. Yang and P. Ashar, *Efficient Distributed SAT and SAT-Based Distributed Bounded Model Checking*, in: *Proc. of CHARME*, 2003, pp. 334–347.

[14] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: *Proceedings of the 8th international SPIN workshop on Model checking of software* (2001), pp. 217–234.

[15] Gupta, A., M. Ganai, C. Wang, Z. Yang and P. Ashar, *Abstraction and BDDs Complement SAT-based BMC in Di*Ver, in: J. Warren A. Hunt and F. Somenzi, editors, *Proc. of the* 15th *Conf. on Computer-Aided Verification*, Lecture Notes in Computer Science **2725** (2003), pp. 206–209.

[16] Gupta, A., M. Ganai, C. Wang, Z. Yang and P. Ashar, *Learning from BDDs in SAT-based Bounded Model Checking*, in: *Proc. of the* 40th *Design Automation Conf.*, 2003, pp. 824–829.

[17] Hazelhurst, S., O. Weissberg, G. Kamhi and L. Fix, *A Hybrid Verification Approach : Getting Deep into the Design*, in: *Proc. of the* 39th *Design Automation Conference*, 2002, pp. 111–116.

[18] Heyman, T., D. Geist, O. Grumberg and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, in: O. Grumberg, editor, *Proc. of Computer Aided Verification*, Lecture Notes in Computer Science **1855** (2000), pp. 20–35.

[19] Ho, P.-H., T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor and J. Long, *Smart Simulation Using Collaborative Formal and Simulation Engines*, in: *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, 2000, pp. 120–126.

[20] Iyer, M. K., G. Parthasarathy and K.-T. Cheng, *SATORI-A Fast Sequential SAT Engine for Circuits*, in: *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, 2003, pp. 320–325.

[21] Jain, J., *On analysis of boolean functions*, Ph.D Dissertation, Dept. of Electrical and Computer Engineering, The University of Texas at Austin (1993).

[22] Kuehlmann, A., V. Paruthi, F. Krohm and M. K. Ganai, *Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification*, IEEE Trans. on CAD **21** (2002), pp. 1377–1394.

[23] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an Efficient SAT Solver*, in: *Proc. of the* 38th *Design Automation Conf.*, 2001, pp. 530–535.

[24] Narayan, A., J. Jain, M. Fujita and A. Sangiovanni-Vincentelli, *Partitioned-ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions*, in: *Proc. of the Intl. Conf. on Computer-Aided Design*, 1996, pp. 547–554.

[25] Ravi, K. and F. Somenzi, *Hints to accelerate symbolic traversal.*, in: *Proc. of CHARME*, 1999, pp. 250–264.

[26] Stern, U. and D. L. Dill, *Parallelizing the murphy verifier*, in: *Proc. of Computer Aided Verification*, Lecture Notes in Computer Science **1254** (1997), pp. 256–267.

[27] Stornetta, T. and F. Brewer, *Implementation of an efficient parallel BDD package*, in: *Proceedings of the 33rd annual conference on Design automation* (1996), pp. 641–644.

[28] Yang, B. and D. R. O'Hallaron, *Parallel breadth-first bdd construction*, in: *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1997), pp. 145–156.

[29] Yang, C. H. and D. Dill, *Validation with Guided Search of the State Space*, in: *Proc. of the* 35th *Design Automation Conference*, 1998, pp. 599–604.

Distributed Symbolic Bounded Property Checking [1]

Pradeep K. Nalla, Roland J. Weiss, Prakash Peranandam,
Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, 72076 Tübingen, Germany

Abstract

In this paper we describe an algorithm for distributed, BDD-based bounded property checking and its implementation in the verification tool SymC. The distributed algorithm verifies larger models and returns results faster than the sequential version.

The core algorithm distributes partitions of the state set to computation nodes after reaching a threshold size. The nodes proceed with image computation on the nodes asynchronously. The main scalability problem of this scheme is the overlap of state set partitions. We present static and dynamic overlap reduction techniques.

Key words: Verification, bounded model checking, property checking, binary decision diagrams, parallelization.

1 Introduction

Although symbolic representations of state spaces [8] based on Binary Decision Diagrams (BDDs) [7] and bounded model checking (BMC) [3] have dramatically increased the design sizes that can be handled by verification tools, research in model checking techniques still concentrates on enabling faster verification of larger models. Large designs cause memory overflow during exploration of the state space, the dreaded state space explosion. There are several proposed solutions to deal with the immense memory requirements of BDDs. One proposal is to partition BDDs [30] into two or more pieces and handle them separately during further traversal. The traversal of the partitions can be done sequentially [10] or in parallel [14].

In [28], a combination of on-the-fly [12] and bounded model checking is presented, which is implemented in the tool SymC. The checking algorithm traverses

[1] This work has been funded in part by the German Research Council (DFG) within projects GRASP and KOMFORT and by the BMBF and edacentrum within project FEST.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the product automaton of model and property until it either detects a validation or a violation of the property, or the explicit or implicit time bound is reached. Only the frontier set is kept in memory, i.e. no fix-point iterations are performed. This approach performs well for certain classes of models and properties, but the sequential version also faces memory exhaustion for large model, e.g. for some of the ISCAS89 examples. This fact motivated the parallelization of the proof algorithm which we present here.

The paper is organized as follows. The next section discusses related work and our contributions. Section 3 summarizes symbolic bounded property checking, followed by a description of the distributed algorithm. Then, we present our static and dynamic methods for overlap reduction. Section 6 gives experimental results. Finally, we conclude and mention future work.

2 Related Work

2.1 Partitioning

Many approaches for decomposing Boolean functions represented as BDDs exist in literature. For distributed verification [16,14] splitting algorithms aim at creating balanced partitions. However, similar approaches exist in sequential verification methodologies [11,10]. The main distinguishing feature of these algorithms is the employed cost function for selecting the splitting variable. The cost functions typically take into account the achieved memory reduction, the amount of sharing between the cofactors, and the memory balance of the cofactors. Also, the CUDD package [31] contains various decomposition algorithms, producing both balanced and unbalanced partitions. Furthermore, decomposition techniques allow representing the same function with multiple BDDs but requiring less memory [20,19]. The image computation algorithms have to be updated for these techniques. The more complex operations are set off by the reduced peak memory requirements of the BDDs [30]. As shown in [4], the reduction can even be exponential. Finally, dense under-approximations [26,25] try to reduce the memory requirements of the BDD but still capture a large percentage of the state space. These algorithms are of minor interest for state set distribution as they result in unbalanced subsets.

None of the proposed heuristics consider subsequent state overlap. However, similar efforts are undertaken for model checkers with an explicit state graph representation [21,5]. They apply graph algorithms that heuristically try to find partitions with few crossover transitions in order to reduce the communication effort between processes. In [15], the authors investigate state space distribution in the context of model checking Petri nets, also employing an explicit representation. These approaches cannot be directly applied to symbolic representations.

2.2 Distributed model checking

The state space explosion problem in model checking has raised interest in handling this problem by adjusting the algorithms for distributed environments recently. This

includes both explicit [32,5,17,6] and symbolic [14,16,2,13,18] model checking methodologies.

The group at Haifa also works on the parallelization of BDD-based verification algorithms. At the core, they create k slices of the current state set and distribute these slices to k cluster machines. They use the slicing technology from [20], but with an enhanced cost function for selecting slicing variables [16]. States are classified as *owned* and *non-owned*. After every image computation step the non-owned states are distributed to the owning nodes. In [16] load balancing is achieved by adjusting the slices if the initial balance is lost. In [14] they try to keep only as many nodes busy as necessary by splitting and joining BDDs on demand. The exchange of non-owned states after every step makes their algorithm mainly synchronous. In [14,16] reachability is computed with fix-point iterations, in [2] regular expressions are used to indicate illegal behavior and μ-calculus formulas are checked in [13]. Our approach checks time-bounded properties specified in PSL (Property Specification Language) [1] or FLTL (Finite Linear Time Temporal Logic) [27] without fixpoint iterations.

2.3 Contributions

Synchronous schemes for parallelizing BDD-based verification algorithms reduce the potential speedup because processes are kept waiting for others to complete. Up to now, no successful asynchronous BDD-based verification algorithms have been proposed.

The main contribution of our approach is such an asynchronous distributed algorithm. This algorithm becomes feasible only when the shared states due to crossover transitions are reduced to avoid duplicate work. We present algorithms for static and dynamic overlap reduction.

3 Sequential Symbolic Bounded Property Checking

The formal verification algorithms in [28,22] combine bounded property checking and symbolic traversal. The temporal logic formulas are converted to special finite state machines called Accept-Reject automata (AR-automata) [27]. AR-automata allow finding violations or validations of properties on finite sequences, thus they are well suited for bounded property checking. The checking algorithm manipulates both the system description and the AR-automata represented as BDDs. In order to avoid the construction of the complete transition relation, a set of conjunctively partitioned transition relations is built, which is used for early quantification [9]. The algorithms have been implemented in the tool SymC, whose general operation is shown in Fig. 1.

An iteration of the sequential verification algorithm works in two steps. First, the successor states of the AR-automata are computed and the termination condition is checked. If the termination condition is not satisfied, image computation is performed on the system in the second step. During image computation the con-

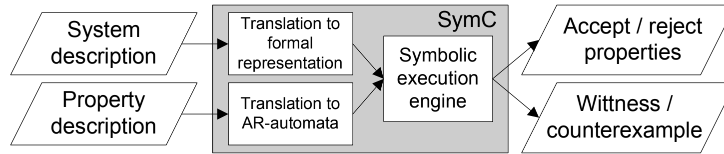


Fig. 1. Overview of SymC operation.

junction of all partitions is built on-the-fly to obtain the successor state set. Like bounded model checking [3], this property checking algorithm does not traverse the state space exhaustively but examines all reachable states within a given time bound.

A central optimization technique for the algorithm is state set splitting. Whenever a threshold for the size of the BDD representing the current state set is reached, the set is split into disjoint parts and the algorithm continues working on these subsets in a divide-and-conquer manner.

The sequential verification algorithm continues with one of the subsets and stacks the others. This can happen recursively. Traversal proceeds on the current subset until the time bound is reached or the termination condition is satisfied. Termination stops the verification with finding either a validation or a violation of the property. Otherwise, the process is repeated for all stacked subsets. The termination condition differs if one checks the property on all paths, i.e. universal quantification, or on one path, i.e. existential quantification. Informally, the sequential termination condition is defined as follows:

Universal If one reject state is detected in the current state set, a violation of the property is found. If all states in the current state set are accepting states, a validation of the property is found. Otherwise, the property is still pending.

Existential If one accept state is detected in the current state set, a validation of the property is found. If all states in the current state set are rejecting states, a violation of the property is found. Otherwise, the property is still pending.

4 Parallelization of Bounded Property Checking

The distributed checking algorithm is composed of an initial sequential stage and a subsequent parallel stage. First, the transition relation is created on all k computation nodes and state space traversal proceeds sequentially on one node until a threshold limit on the BDD size triggers state set distribution. The splitting into k subsets is already performed in parallel and every node is responsible for getting its own disjoint part of the whole state set. The nodes start state space traversal independently on these subsets. The termination condition stays the same, however the nodes have to communicate their local results in order to allow testing termination conditions that depend on all states.

This simple scheme fails to provide significant speedups on many models because of crossover transitions. These transitions start in a state of the current subset but lead to a state that is already present in one of the other state subsets. We call

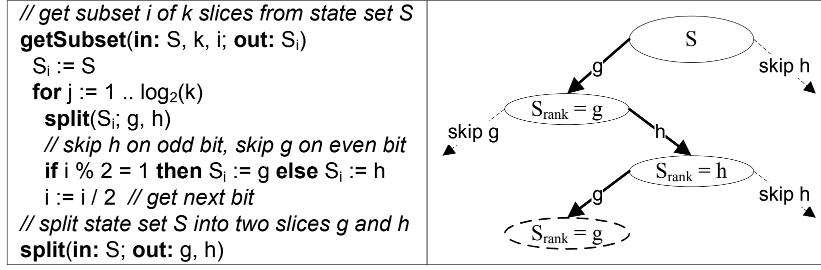


Fig. 2. Algorithm for state set distribution. The left hand side gives the distribution algorithms, an example application is shown on the right hand side.

this phenomenon state set overlap, or just overlap. Of course, image computation for overlapping states is performed redundantly. As image computation is one of the key components of formal verification tools, redundancy of such a component badly affects the time and memory requirements of the whole verification process. Thus, optimizing the distributed algorithm concentrates on reducing the overlap (see section 5).

4.1 State set distribution

Splitting the state set into k parts for subsequent traversal in parallel is a costly operation. Therefore, we already perform it in parallel. For simplicity we assume that $k = 2^n, n \in \mathbb{N}$. Basically, once the first node dumps its state set to disk, all other nodes pick up the dumped set after notification. Then, each node splits the set into two parts and depending on its rank, a number identifying every node, it drops one part and continues splitting on the other part recursively until only its own subset remains. The algorithm is illustrated in Fig. 2.

4.2 State set overlap

After all nodes picked their state subsets, the nodes proceed with symbolic state space traversal. A very important observation is that after a few steps of traversal state overlap between network nodes may emerge.

Definition 1 Let S be a set represented using a BDD. Then $\|S\|$ denotes the number of states in S, which is given by the number of maximal minterms of the BDD.

Definition 2 Let S be a nonempty set and $S_1, \ldots, S_k \subseteq S$ with $k \geq 2$. Then we define the state overlap $o_k \in [0, 1]$ of these partitions as:

$$o_k = \frac{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \|S_i \cap S_j\|}{\|S\| \sum_{i=1}^{k-1} i}. \tag{1}$$

The overlap is thus the normalized average of states in the pairwise intersection of subset permutations. The sum in the denominator ranges from 1 to $k-1$ because this yields the number of pairs S_i, S_j with $i < j$. An overlap of $o_k = 0$ corresponds

to disjoint partitions and an overlap of $o_k = 1$ corresponds to partitions containing the same states.

5 Overlap Reduction

Boolean functions represent all the state sets and the transition relation in symbolic traversal. This representation can grow large if the sets to be represented are big, corresponding directly to more memory requirements. The Boolean functions are represented and manipulated using BDDs. The memory requirements $|f|$ of a Boolean function f are defined as the number of its nodes. In order to reduce the memory requirements one can partition a Boolean function into smaller parts, whose union is the whole set.

Definition 3 Given a Boolean function $f : B^n \to B$, f is partitioned into two functions f_1 and f_2 on a variable v from the support set of f with

$$f = f_1 \vee f_2 \text{ where } f_1 = v \wedge f_v,\ f_2 = \bar{v} \wedge f_{\bar{v}}. \quad (2)$$

The *splitting variable* v defines the partitioning of f into f_1 and f_2. This splitting can be implemented easily with BDD operations. BDDs are compressed decision trees where common subtrees are joined. This causes significant sharing of nodes in a function's representation. Thus, splitting a function f into two functions f_1 and f_2 with a poor choice of v may not necessarily reduce the memory requirements of the split functions and can result in $|f_1| \approx |f_2| \approx |f|$. In the following discourse, we identify a state set with its characteristic function represented as BDD.

5.1 Static overlap reduction

Overlap originates from states in different sets having transitions to the same next states. In order to minimize the overlap of splits, the selected splitting variable v should not allow states that have common next states to be in different splits. In other words, v should partition the states such that they have no common next states. However, in reality such a partitioning is not possible, but one can put some effort in selecting the splitting variable v to minimize overlap. For finding a good splitting variable we statically analyze the design which is represented as finite state machine (FSM).

Definition 4 A FSM $\mathcal{A}$ is a 4-tuple $\mathcal{A} = (S, \Sigma, \mathcal{T}, \mathcal{I})$, where $S = \{s_1, \ldots, s_n\}$ is a finite set of states encoded by state variables $e_1, \ldots, e_m$, Σ is a finite input alphabet, $\mathcal{T} \subseteq S \times \Sigma \times S$ is a transition relation represented with $T_1, \ldots, T_m$ partitions, and $I \subseteq S$ is the set of initial states.

The idea of selecting a good splitting variable v relies on the conjunctively partitioned transition relation $\mathcal{T}$ [9]. For every $i \in 1, \ldots, m$ a partition T_i of the transition relation corresponds to the truth value of next state variable e'_i such that

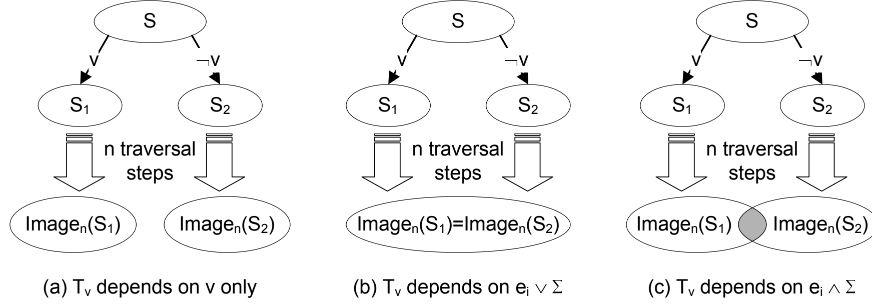


Fig. 3. Possible overlap of subsets after n steps with dependencies on the splitting variable.

$\mathcal{T} = \bigwedge_{i=1}^{m} T_i$. We pick v from the set of state variables $E = \{e_1, \ldots, e_m\}$. Fig. 3 (a) shows the best case where there is no overlap. This kind of situation is only possible if v will stick to its truth value in all further steps, i.e. partition T_v (T_i *with* $v = e_i$) depends only on v. Though this is the ideal case, we hardly have such situations in real designs. This means that v might change its truth value in future steps as its partition of the transition relation depends on more factors. The worst case of almost complete overlap can occur if T_v depends on input variables disjunctively only, as depicted by Fig. 3 (b). The common case lies in between these two extremes and happens when v depends on inputs conjunctively with other combinations of state variables, depicted in Fig. 3 (c). The algorithm *MinOverlap* pioneers in exploiting static information of the partitioned transition relation $\mathcal{T}$ to find a good splitting variable v.

In a pre-processing step, every state variable is assigned an *influence* and the variables are ordered decreasingly by their influence. The influence table maps state variables to their influence. Later, the splitting variable selection algorithm utilizes this information.

Definition 5 Let $l_1, l_2 \in \mathbb{N}$ be influence lookaheads. For a given FSM $\mathcal{A}$, the influence $\Phi_{l_1,l_2}(e) \in [-1, 1]$ of a state variable $e \in E$, with $|E| = m$, is defined as

$$\Phi_{l_1,l_2}(e) = \frac{|\mathcal{D}^{\uparrow}(e, l_1)| - |\mathcal{D}^{\downarrow}(e, l_2)|}{m}. \tag{3}$$

Set $\mathcal{D}^{\uparrow}(e, l_1)$ contains all state variables that get influenced by e in l_1 steps, and set $\mathcal{D}^{\downarrow}(e, l_2)$ contains all state variables that influence e in l_2 steps. These sets are determined iteratively starting with $l_1 = 1$ and $l_2 = 1$. Each T_i directly corresponds to the truth value of the next state variable e'_i, so we compute these sets by walking all T_i and e_i. For $\mathcal{D}^{\uparrow}(e, 1)$, we count the partitions T_i that contain e, whereas for $\mathcal{D}^{\downarrow}(e, 1)$ we count the state variables in the support of T_i.

The basic assumption of the *MinOverlap* algorithm is that splitting on a variable v with high influence will lead to fewer cross transitions between the resulting partitions, because the value of $\Phi_{l_1,l_2}(v)$ next state variables depends on v. Of course, there are other factors determining the values of these next state variables, weakening our assumption. Our algorithm works well if the partitioned transition relations T_i depend on conjunctively connected variables only. It degrades if the T_i depend

on disjunctively connected variables where at least one disjunct contains only input variables. However, it is computationally expensive to analyze all Boolean connectives of the clauses of every T_i.

The actual *MinOverlap* algorithm picks a viable state variable for splitting. The state variables are categorized based on their influence and put into different sets. We start with the set containing variables with a high influence and check them against a balancing condition. Alongside, we compute the cost of these variables with the cost function from [16] that consists of a redundancy and a reduction factor. If none of the examined variables satisfied the balancing condition, the variable with minimal cost is selected. Fig. 4 gives the pseudo code for *MinOverlap*.

```
// S is the current state set                                  1
// S1 and S2 are the resulting partitions                      2
// Φ is the influence table                                    3
// δ is the memory balance factor                              4
// α is the weight for the cost function                       5
split(in: S, Φ, δ, α; out: S1, S2)                             6
 bestCost := Φ.top()                                           7
 minCost := cost(S, bestCost, δ, α)                            8
 while C = getCandidateSet(Φ) ∧ C ≠ ∅                          9
  for all w ∈ C                                                10
   if max(|S_w|, |S_w'|) ≤ δ|S| then                           11
    v := w; goto do_split                                      12
   else                                                        13
    thisCost := cost(S, v, α)                                  14
    if thisCost < minCost then                                 15
     minCost := thisCost; bestCost := w                        16
 v := bestCost                                                 17
 do_split: S1 := S_v; S2 := S_v'                               18
```

Fig. 4. State set splitting with the *MinOverlap* algorithm.

5.2 Dynamic overlap reduction

Initially, the overlap between state sets of network nodes is reduced by applying the *MinOverlap* algorithm. However, in general the overlap may still pursue after a few steps of state space traversal. In order to further confine the overlap we perform dynamic overlap reduction. This is a methodology where we allow overlap to some extent and heuristically select a time frame to remove it periodically. We perform overlap removal after state set distribution (see section 4.1). This method is iteratively performed either throughout the verification process or up to n times. An extra node called *coordinator* organizes the communication between the nodes and performs dynamic removal of state overlap. The overlap removal algorithm for each node works in three steps:

(i) Upon reaching a reduction time point[2] the node dumps its current state set onto the network drive and sends a message to the coordinator.

(ii) The coordinator removes the overlap of the node with respect to the already visited state space by other nodes at this time point, and updates the history of the visited state space. Then it informs the corresponding node to proceed with the reduced state space by dumping the trimmed state set.

(iii) Finally, if all nodes passed a reduction time point, the coordinator removes the state space history of that time point.

Fig. 5 delineates the usage of overlap reduction in the main computation loop of the symbolic simulation algorithm in the parallel stage. We have to check the termination condition locally, i.e. only in the current subset (line 10), and globally, which requires communication with the other nodes (line 8). For example, in order to show an universal validation, all nodes have to finish in accept states locally, which can only be checked globally.

```
// S is the set of initial states                                            1
// t is the checking time bound                                              2
// p is the period of steps at which overlap removal is performed            3
// n is the overlap removal limit, 0 indicates continuous reduction          4
simulate(in: S, t, p, n)                                                     5
 reduction_limit := 0; reduction_step := 0                                   6
 if n > 0 then tillEnd := false else tillEnd := true                         7
 while iteration < t                                                         8
  checkTerminationConditionGlobally()                                        9
  S := image_AR(S) // Compute image of AR-automata.                          10
  checkTerminationConditionLocally(S);                                       11
  S := image_T(S) // Compute image of the system.                            12
  if (reduction_limit < n) ∨ tillEnd then                                    13
   reduction_step++                                                          14
   if reduction_step = p then                                                15
    S := removeOverlap(S)                                                    16
    reduction_step := 0; reduction_limit++                                   17
```

Fig. 5. Main computation loop for state overlap removal.

The main advantage of our dynamic reduction method is that nodes do not have to wait for slow nodes. After dumping their current state set, faster nodes can continue to traverse the product automaton. Therefore, we achieve asynchronous overlap removal between network nodes. Although nodes have to wait for the coordinator to update their state set, this time is not significant compared to the time spent on image computation.

An interesting side effect of our asynchronous methodology is the resulting natural load balancing. The very last node that reaches a reduction time point

[2] The state set distribution time point and the reduction period determine the reduction time points.

gets its overlap removed with respect to all other nodes. So this last node has no states in common with the other nodes at this reduction step. Our experiments state that usually the last node after overlap removal has the smallest subset. This in turn means faster image computation enabling this node to reach the forthcoming reduction time point faster. Hence, at that reduction time point this particular node will arrive earlier than other nodes, and therefore continues with a larger state set. This process alternates among the nodes accordingly depending on the weight of image computation, resulting in natural load balancing between the network nodes.

For some models, the overlap is so high that the late nodes become empty after overlap removal. This special situation is handled by state set sharing with the following node that reaches any reduction time point.

6 Experimental Results

We performed our experiments on the Kepler cluster at the University of Tuebingen [3] . This cluster contains 98 computing nodes, each consisting of dual 650 MHz Pentium-III processors with 1 GB of shared memory (512 MB for each processor). We conducted our experiments on some of the circuits from the ISCAS89 benchmarks and a model of a holonic production system [29]. All experiments were performed with dynamic variable ordering disabled in the BDD package. For circuits from the ISCAS89 benchmarks we check for reachability of a state at high hamming distance from the initial states (see equation 4) along with properties from [2]. In the holonic production system we check for consumption of a workpiece (see equation 5). All properties are checked universally. The properties written in FLTL look like this, where $b > 0$ are explicit time bounds on the properties:

$$\mathsf{G}[b]\ !(\text{s1512.start \& s1512.video \& ... \& s1512.I1733}) \tag{4}$$

$$\mathsf{F}[b]\ \text{OutBuffer.s_consume} \tag{5}$$

6.1 Static overlap reduction

In this part we concentrate on comparing the static overlap reduction heuristic *MinOverlap* to an altered version of the slicing heuristic from [16] labeled *EqualDist*, and the variable disjunction decomposition algorithm from the CUDD package [31] labeled as *VarDisj*. The *MinOverlap* algorithm is denoted by the influence Φ_{l_1,l_2} used for ordering the state variables. In these experiments, we use a balancing condition of $max(|f_1|, |f_2|) \leq \frac{2}{3}|f|$ for the *MinOverlap* algorithm. The results are shown in Fig. 6.

Discussion: The preprocessing step of the *MinOverlap* algorithm does not require a significant amount of time, in all experiments it consumed less than 1% of

[3] http://kepler.sfb382-zdv.uni-tuebingen.de

Design	Split. alg.	Recursion level : Splitting vars.	Step : $o_k \cdot 100$	s_t	v_t
s1269	$\Phi_{1,1}$	1:32	1:31.5	0.11	214.13 (0.55)
#2	$\Phi_{1,0}$	1:32	1:31.5	0.11	217.5 (0.51)
37	EqualDist	1:0	1:58.6	0.36	236.7 (0.28)
5000	VarDisj	1:14	1:57.4	0.15	371.2 (0.25)
s1512	$\Phi_{5,1}$	1:96	5:64.8 / 10:89.3	0.15	3204.85 (4.52)
#2	$\Phi_{1,0}$	1:10	5:89.3 / 10:91.8	0.11	3330.31 (4.4)
57	EqualDist	1:10	5:89.3 / 10:91.8	0.87	3324.89 (3.72)
10000	VarDisj	1:10	5:89.3 / 10:91.8	0.38	3312.75 (3.75)
s1269	$\Phi_{1,1}$	1:32 / 2:34 / 3:36	1:9.1	0.68	69.9 (0.55)
#8	$\Phi_{1,0}$	1:32 / 2:34 / 3:40,36	1:9.1	0.68	69.43 (0.52)
37	EqualDist	1:0 / 2:42,6 / 3:12,8,40	1:12.7	0.55	58.8 (0.29)
5000	VarDisj	1:14 / 2:12 / 3:10,26	NA	0.26	58.4 (0.27)
s1512	$\Phi_{5,1}$	1:96 / 2:98 / 3:100,10	10:54.6 / 15:68.3	0.23	2891.7 (4.6)
#8	$\Phi_{1,0}$	1:10 / 2:12 / 3:14	10:76.4 / 15:94.2	0.15	2993.0 (4.4)
57	EqualDist	1:10 / 2:96,12 / 3:12,14,94	10:76.6 / 15:94.2	1.5	2988.9 (3.7)
10000	VarDisj	1:10 / 2:12,94,96 / 3:12,96,14	10:76.4 / 15:94.2	0.71	3029.2 (3.7)
nh2	$\Phi_{1,1}$	1:14 / 2:48,16 / 3:10,18,52	60:11.7 / 100:25.4	0.65	#481 (7.12)
#8	$\Phi_{1,0}$	1:14 / 2:54,16 / 3:48,18,10	60:12.2 / 100:26.6	0.51	#272 (7.0)
118	EqualDist	1:4 / 2:58,176, / 3:24,134,54,40	60:22.1 / 100:40.8	13.4	#151 (6.04)
50000	VarDisj	1:4 / 2:58,40 / 3:44,54,18,176	60:20.9 / 100:41.0	13.3	#145 (6.07)

Fig. 6. Comparison of *MinOverlap* with other heuristics. The first column lists the design, followed by the number of processors used, the number of state variables and the splitting threshold. The second column indicates the splitting algorithm. The third column gives at each splitting recursion level the indexes of the selected splitting variables. The CUDD package identifies variables by index. Then the fourth column shows the overlap at different iteration steps. The fifth and sixth columns list the average splitting time s_t, and the total verification time v_t (or a memory overflow is indicated by #, followed by the maximum number of steps), respectively. The splitting time corresponds to the time spent in algorithm split as described in figure 4.

the verification time. For two processor, design *s1269* shows a significant reduction in overlap by selecting high influence variables and hence a gain in overall verification time can be observed. Both *MinOverlap* and *EqualDist* picked high influence variables, but only *MinOverlap* reduced the overlap significantly. This is due to the influence lookahead condition explained in Section 5.1. The influence stays positive for *MinOverlap* and becomes negative for *EqualDist* with $\Phi_{1,1}$.

For eight processors, design *s1269* has low overlap with all splitting algorithms. But the other two designs clearly show the benefit of applying *MinOverlap*, both for designs with huge and moderate overlap. Design *s1512* belongs to the category with huge overlap after a few steps. However, *MinOverlap* with a lookahead of $l_1 = 5$ is able to significantly delay the occurrence of overlap and reduce the verification time. Nevertheless, this reveals that high influence variables can only help to reduce the overlap for a few steps but cannot avoid it beyond a limit, making dynamic removal techniques a must. The overlap in design *nh2* increases much slower than

in the other design, but its size leads to memory overflow. Again, *MinOverlap* is able to reduce the overlap, even after 100 steps. This allows the nodes to go a lot further without memory overflow.

6.2 Dynamic overlap reduction

First, we ran some of the larger designs in sequential SymC with all relevant optimizations switched on. The sequential algorithm splits the state set repeatedly upon reaching the threshold, whereas the parallel version does it only during state set distribution. The results are available in Fig. 7. For most of the designs the sequential algorithm cannot complete traversal due to memory overflow or time out problems [4] .

Design	Threshold	Φ_{l_1,l_2}	**Time bound**	**Peak node count**	v_t
s4863	20000	$\Phi_{1,1}$	5	4.48	#2
s1512	50000	$\Phi_{2,1}$	100	3.80	*80
s1423$_{p1}$	50000	$\Phi_{1,0}$	-	13.55	#11
s1423$_{p3}$	50000	$\Phi_{1,0}$	12	14.27	*11
nh2	50000	$\Phi_{1,1}$	1000	2.30	663

Fig. 7. Results for fully optimized sequential SymC. The first column lists the design name. Column two gives the splitting threshold. The third column shows the influence used for *MinOverlap* splitting. The fourth and fifth columns list the time bound specified in the property and maximum peak node count in millions, respectively. The last column shows the overall verification time. #n or *n denote memory overflow or time out at step n.

Fig. 8 shows the results of the distributed approach with dynamic overlap removal using 32 processors dedicated to the checking algorithm and one processor acting as the coordinator. In these experiments, dynamic overlap removal is applied throughout the verification process repeatedly every p steps.

Discussion: First of all, the parallel algorithm is able to finish all the problems that the sequential approach was not able to handle due to space or time restrictions.

Designs *s4863* and *s1512* clearly show the advantage of both parallelization and dynamic overlap removal, i.e. decreasing p reduces verification time. Also, traversal of design *nh2* completes with a speedup of 2.8 compared to the sequential version.

For design *s1423* we considered three properties $p1$, $p2$ and $p3$. Both $p1$ and $p2$ are from [2] and pure LTL properties, hence there is no time bound specified in the property. In comparison to [2], SymC finds errors in the designs significantly faster, even taking different hardware configurations into account. However, design *s1423* behaves unexpectedly as verification time increases with shorter dynamic overlap reduction periods. This effect is caused by the behavior of the BDDs representing the state sets. Removing states from the sets actually increases their

[4] Experiments were stopped after one hour.

Design	p (Φ_{l_1,l_2})	Time bound	Seq. time (step)	Peak node count	v_t
s4863	1 ($\Phi_{1,1}$)	5	1.67 (1)	8.39	587.73
20000	2 ($\Phi_{2,1}$)	5	1.69 (1)	10.52	613.32
s1512	2 ($\Phi_{2,1}$)	100	108.75 (33)	2.41	508.21
10000	3 ($\Phi_{3,1}$)	100	108.95 (33)	2.55	522.25
	5 ($\Phi_{5,1}$)	100	106.53 (33)	2.83	643.61
s1423$_{p1}$	1 ($\Phi_{1,1}$)	-	75.70 (8)	13.20	748.5
50000	2 ($\Phi_{2,1}$)	-	75.54 (8)	13.77	806.99
	5 ($\Phi_{5,1}$)	-	76.13 (8)	11.23	567.41
s1423$_{p2}$ 50000	1 ($\Phi_{1,1}$)	-	76.62 (8)	1.87	114.25
s1423$_{p3}$	1 ($\Phi_{1,1}$)	12	152.04 (9)	14.51	1322.22
50000	2 ($\Phi_{2,1}$)	12	151.84 (9)	13.18	1171.18
	3 ($\Phi_{3,1}$)	12	153.18 (9)	7.68	791
nh2 50000	100 ($\Phi_{1,1}$)	1000	86.22 (132)	1.658	230.08

Fig. 8. Results of the distributed algorithm with dynamic overlap removal. The first column indicates the design and the splitting threshold. The second column shows the time period p at which overlap reduction is performed and the influence used in *MinOverlap*. The third column lists the time bound specified in the property. Column four lists the time taken by the sequential part and the time step at which the parallel stage starts. Column five shows the maximum peak node count of all the nodes in millions. The last column lists the overall verification time.

BDD representation. This opens a new thread for heuristics when and how to apply dynamic removal. We also investigate if dynamic variable reordering takes care of this problem.

Fig. 9 depicts the natural load balance graph for the circuit *s1512* with reduction period 2. Only four nodes are shown for clear visibility of the graph. The load balancing effect can be seen very well when nodes 0 and 24 swap their arrival order during execution.

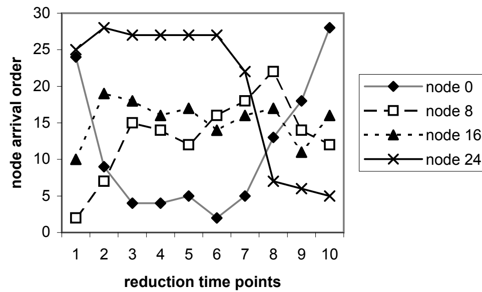


Fig. 9. Arrival order of nodes at reduction points showing load balancing between the nodes.

Finally, measurements indicate that reading and writing BDDs to and from disk does not contribute to the overall verification time significantly. Thus, network I/O is not a bottleneck of the distributed algorithm.

7 Conclusions and Future Work

This paper presents the parallelization of a BDD-based bounded property checking algorithm. The two main contributions are a novel splitting algorithm taking overlap reduction into account and a distributed on-the-fly algorithm for asynchronous state space traversal with dynamic overlap reduction resulting in natural load balancing.

The *MinOverlap* splitting heuristic enhances current decomposition algorithms by preprocessing the transition relation and using this information for ordering the list of potential splitting variables. The experiments show that this preprocessing step is able to actually reduce the overlap and the splitting time. Furthermore, *MinOverlap* almost never degrades the splitting runtime or the resulting overlap significantly.

Dynamic overlap reduction is an important technique in enabling verification of larger designs and significantly improves the applicability of the distributed algorithm. Reassigning idle nodes avoids wasted computation power. However, for some designs overlap reduction can actually increase the BDD representation of sets with fewer states. This seems to be related to the characteristic that a fixed BDD variable order is kept after the sequential stage. We experiment with different variable orderings on computation nodes to handle these cases. Furthermore, we are extending our experiments to designs from the VIS suite and recent IBM examples.

8 Acknowledgements

We want to thank the reviewers for their detailed comments that helped in enhancing the quality of this paper.

References

[1] Accellera, "Property Specification Language (PSL), Version 1.1," (2004), http://www.eda.org/vfv.

[2] Ben-David, S., T. Heyman, O. Grumberg and A. Schuster, *Scalable distributed on-the-fly symbolic model checking*, International Journal on Software Tools for Technology Transfer (STTT) **4(4)** (2003), pp. 496–504.

[3] Biere, A., A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu, *Bounded model checking*, in: M. Zelkowitz, editor, *Highly Dependable Software*, Advances in Computers **58**, Academic Press, 2003 .

[4] Bollig, B. and I. Wegener, *Partitioned BDDs vs. other BDD models*, in: *ACM/IEEE International Workshop on Logic Synthesis (IWLS)*, 1997.

[5] Braberman, V., A. Olivero and F. Schapachnik, *Issues in distributed timed model checking: Building Zeus*, International Journal on Software Tools for Technology Transfer (STTT) **7(1)** (2005), pp. 4 – 18.

[6] Brim, L., I. Černá, P. Moravec and J. Šimša, *Distributed partial order reduction of state spaces*, in: Proceedings of PDMC 2004 [24].

[7] Bryant, R. E., *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Computing Surveys **24(3)** (1992), pp. 293–318.

[8] Burch, J., E. Clarke, K. L. McMillan, D. Dill and L. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, Information and Computing **98** (1992), pp. 142–170.

[9] Burch, J. R., E. M. Clarke and D. E. Long, *Representing circuits more efficiently in symbolic model checking*, in: *28th Conference on Design Automation* (1991), pp. 403–407.

[10] Cabodi, G., P. Camurati, L. Lavagno and S. Quer, *Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits*, in: *34th Conference on Design Automation* (1997), pp. 728–733.

[11] Cabodi, G., P. Camurati and S. Quer, *Improved reachability analysis of large finite state machines*, in: Proceedings of ICCAD 1996 [23], pp. 354–360.

[12] Clarke, E. M., O. Grumberg and D. E. Peled, "Model Checking," The MIT Press, 1999.

[13] Grumberg, O., T. Heyman and A. Schuster, *Distributed symbolic model checking for* μ*-calculus*, in: G. Berry, H. Comon and A. Finkel, editors, *Computer Aided Verification, 13th International Conference*, Lecture Notes in Computer Science **2102** (2001), pp. 350–362.

[14] Grumberg, O., T. Heyman and A. Schuster, *A work-efficient distributed algorithm for reachability analysis*, in: W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference*, Lecture Notes in Computer Science **2725** (2003), pp. 54–66.

[15] Haverkort, B., A. Bell and H. Bohnenkamp, *On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets*, in: *8th International Workshop on Petri Nets and Performance Models* (1999).

[16] Heyman, T., 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, *Computer Aided Verification, 12th International Conference*, Lecture Notes in Computer Science **1855** (2000), pp. 20–35.

[17] Inggs, C. P. and H. Barringer, *CTL* model checking on a shared-memory architecture*, in: Proceedings of PDMC 2004 [24].

[18] Lange, M. and H. W. Loidl, *Parallel and symbolic model checking for fixpoint logic with Chop*, in: Proceedings of PDMC 2004 [24].

[19] McMillan, K. L., *A conjunctively decomposed boolean representation for symbolic model checking*, in: R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference*, Lecture Notes in Computer Science **1102** (1996), pp. 13–25.

[20] Narayan, A., J. Jain, M. Fujita and A. L. Sangiovanni-Vincentelli, *Partitioned ROBDDs - a compact, canonical and efficiently manipulable representation for boolean functions*, in: Proceedings of ICCAD 1996 [23], pp. 547–554.

[21] Orzan, S., J. van de Pol and M. V. Espada, *A state space distribution policy based on abstract interpretation*, in: Proceedings of PDMC 2004 [24].

[22] Peranandam, P. M., R. J. Weiss, J. Ruf, T. Kropf and W. Rosenstiel, *Dynamic guiding of bounded property checking*, in: *IEEE International High Level Design Validation and Test Workshop 2004 (HLDVT 04)*, 2004.

[23] "Proceedings of ICCAD 1996," ACM and IEEE Computer Society Press, 1996.

[24] "Proceedings of PDMC 2004," Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[25] Ravi, K., K. L. McMillan, T. R. Shiple and F. Somenzi, *Approximation and decomposition of binary decision diagrams*, in: *35th Conference on Design Automation* (1998), pp. 445–450.

[26] Ravi, K. and F. Somenzi, *High-density reachability analysis*, in: *1995 IEEE/ACM International Conference on CAD* (1995), pp. 154–158.

[27] Ruf, J., D. W. Hoffmann, T. Kropf and W. Rosenstiel, *Simulation-guided property checking based on a multi-valued AR-automata*, in: W. Nebel and A. Jerraya, editors, *Design, Automation and Test in Europe 2001* (2001), pp. 742–748.

[28] Ruf, J., P. M. Peranandam, T. Kropf and W. Rosenstiel, *Bounded property checking with symbolic simulation*, in: *Forum on Specification and Design Languages 2003*, 2003.

[29] Ruf, J., R. J. Weiss, T. Kropf and W. Rosenstiel, *Modeling and formal verification of production automation systems*, in: E. et. al., editor, *Integration of Software Specification Techniques for Applications in Engineering*, Lecture Notes in Computer Science **3147**, Springer, 2004 pp. 541–566.

[30] Sahoo, D., S. K. Iyer, J. Jain, C. Stangier, A. Narayan, D. L. Dill and E. A. Emerson, *A partitioning methodology for BDD-based verification*, in: A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design, Fifth International Conference*, Lecture Notes in Computer Science **3312** (2004), pp. 399–413.

[31] Somenzi, F., *CUDD: CU decision diagram package, release 2.4.0*, http://vlsi.colorado.edu/~fabio/CUDD (2004).

[32] Stern, U. and D. L. Dill, *Parallelizing the Murϕ verifier*, in: O. Grumberg, editor, *Computer Aided Verification, 9th International Conference*, Lecture Notes in Computer Science **1254** (1997), pp. 256–278.

A Pattern Recognition Approach for Speculative Firing Prediction in Distributed Saturation State-Space Generation

Ming-Ying Chung [1] and Gianfranco Ciardo [2]

University of California, Riverside, CA 92521

Abstract

The *saturation* strategy for symbolic state-space generation is particularly effective for globally-asynchronous locally-synchronous systems. A distributed version of saturation, *SaturationNOW*, uses the overall memory available on a network of workstations to effectively spread the memory load, but its execution is essentially sequential. To achieve true parallelism, we explore a *speculative firing prediction*, where idle workstations work on predicted future event firing requests. A naïve approach where all possible firings may be explored a priori, given enough idle time, can result in excessive memory requirements. Thus, we introduce a history-based approach for firing prediction that recognizes firing patterns and explores only firings conforming to these patterns. Experiments show that our heuristic improves the runtime and has a small memory overhead.

Keywords: state-space generation, decision diagrams, distributed systems, parallel and distributed computing, speculative computing, pattern recognition

1 Introduction

Formal verification techniques such as model checking [10] are widely used in industry for quality assurance, since they can be used to detect design errors early in the lifecycle. An essential step is an exhaustive, and very memory-intensive, state-space generation. Even though symbolic encodings like binary decision diagrams (BDDs) [2] help cope with the *state-space explosion*, the analysis of complex systems may still resort to the use of virtual memory.

Much research has then focused on parallel and distributed computing for this application. [1,20,25] use a network of workstations (NOW) for *explicit*

[⋆] Work supported in part by the National Science Foundation (NSF) under Grants No. 0219745 and No. 0203971.

[1] Email:chung@cs.ucr.edu

[2] Email:ciardo@cs.ucr.edu

state-space exploration or model checking. [16] parallelizes BDD manipulation on a shared memory multiprocessor, while [21] uses distributed shared memory. [27] parallelizes BDD construction by sharing image computation among processors during Shannon expansion on shared and distributed shared memory platforms. [11] finds parallelism in breadth-first BDD traversals. [13,17,26] parallelize BDD manipulations by slicing image computation onto a NOW where a master workstation balances the memory load.

In [4], we presented a distributed version of the saturation algorithm [6], called *SaturationNOW*, to perform symbolic state-space generation on a NOW, where execution is strictly sequential but utilizes the overall NOW memory. As in [23], a *level-based horizontal "slicing"* scheme is employed to allocate decision diagram nodes to workstations, so that no additional node or work is created. In addition, we presented a heuristic that dynamically balances the memory load to help cope with the changing peak memory requirement of each workstation. However, the horizontal slicing scheme has two drawbacks. First, while it can evenly distribute the decision diagram with minimal time and space overhead, it does not facilitate parallelism (it corresponds to a sequentialization of the workstations, where most computations require a workstation to cooperate with its neighbors). Second, since a set of contiguous decision diagram levels is assigned to each workstation, models with few decision diagram levels impose a limit on the scalability of the approach. While assigning a single level to multiple workstations solves this problem, the cost of additional synchronizations would eliminate the major advantage of our horizontal slicing scheme.

In this paper, we tackle the first drawback, i.e., we improve the runtime of SaturationNOW, through the idea of using idle workstation time to speculatively fire events on decision diagram nodes, even if some of these event firings may never be needed. In a naïve approach, unrestrained speculation may cause an excessive increase in the memory consumption, to the point of being counter-productive. However, a history-based approach to predict which events should be fired based on past firing patterns is instead effective at reducing the runtime with only a small memory overhead.

Our paper is organized as follows. Sect. 2 gives background on reachability analysis, decision diagrams, Kronecker encoding, and saturation. Sect. 3 details our naïve and pattern recognition approaches to speculative firing prediction. Sect. 4 shows experimental results. Sect. 5 briefly survey related work, and Sect. 6 discusses future research directions.

2 Background

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^{init}, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^{init} \in \widehat{\mathcal{S}}$ is the *initial state*, and $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in a single step. Since we target globally-asynchronous systems, we decompose $\mathcal{N}$ into a

disjunction of next-state functions [15]: $\mathcal{N}(\mathbf{i}) = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function associated with event e.

The *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ is the smallest set containing $\mathbf{s}^{init}$ and closed with respect to $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}^{init}\} \cup \mathcal{N}(\mathbf{s}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^{init})) \cup \cdots = \mathcal{N}^*(\mathbf{s}^{init})$, where "*" denotes reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}(\mathbf{i})$.

We assume a model composed of K *submodels*. Thus, a (*global*) state is a K-tuple $(\mathbf{i}_K, ..., \mathbf{i}_1)$, where $\mathbf{i}_k$ is the *local* state of submodel k, $K \geq k \geq 1$, and $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, the cross-product of K *local state spaces*. This allows us to use techniques targeted at exploiting system structure, in particular, *symbolic* techniques to store the state-space based on decision diagrams.

2.1 Symbolic encoding of the state space $\mathcal{S}$ and next-state function $\mathcal{N}$

While not a requirement (the local state spaces $\mathcal{S}_k$ can be generated "on-the-fly" by interleaving symbolic global state-space generation with explicit local state-space generation [7]), we assume that each $\mathcal{S}_k$ is known a priori. We then use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, ..., n_k{-}1\}$, with $n_k = |\mathcal{S}_k|$, identify local state $\mathbf{i}_k$ with its index $i_k = \psi_k(\mathbf{i}_k)$, thus $\mathcal{S}_k$ with $\{0, 1, ..., n_k{-}1\}$, and encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ in a *quasi-reduced ordered multiway decision diagram* (MDD) over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed acyclic edge-labeled multi-graph where:

- Each node p belongs to a *level* in $\{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node r at level K.
- Level 0 can only contain the two *terminal* nodes *Zero* and *One*.
- A node p at level $k > 0$ has n_k outgoing edges, labeled from 0 to $n_k{-}1$. The edge labeled by i_k points to a node q at level $k{-}1$; we write $p[i_k] = q$.
- Given nodes p and q at level k, if $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$.

The MDD encodes a set of states $\mathcal{B}(r)$, defined by the recursive formula: $\mathcal{B}(p) = \bigcup_{i_k\in\mathcal{S}_k} \{i_k\} \times \mathcal{B}(p[i_k])$ if $p.lvl = k > 1$, $\mathcal{B}(p) = \{i_1 : p[i_1] = One\}$ if $p.lvl = 1$.

To adopt a Kronecker representation of $\mathcal{N}$ inspired by work on Markov chains [3], we assume a *Kronecker consistent* model [5,6] where $\mathcal{N}_e$ is conjunctively decomposed into K local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying $\forall (i_K, ..., i_1) \in \widehat{\mathcal{S}}, \mathcal{N}_e(i_K, ..., i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$. By defining $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0, 1\}^{n_k \times n_k}$, with $\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,e}(i_k)$, we encode $\mathcal{N}_e$ as a (boolean) Kronecker product: $\mathbf{j} \in \mathcal{N}_e(\mathbf{i}) \Leftrightarrow \bigotimes_{K\geq k\geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1$, where a state $\mathbf{i}$ is interpreted as a *mixed-based* index in $\widehat{\mathcal{S}}$ and "$\bigotimes$" indicates the Kronecker product of matrices. Note that the $\mathbf{N}_{k,e}$ matrices are extremely sparse: for standard Petri nets, each row contains at most one nonzero entry.

2.2 Saturation-based iteration strategy

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognize and exploit *event locality* [5] and employ *saturation* [6]. We say that event e is *independent* of level k if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$

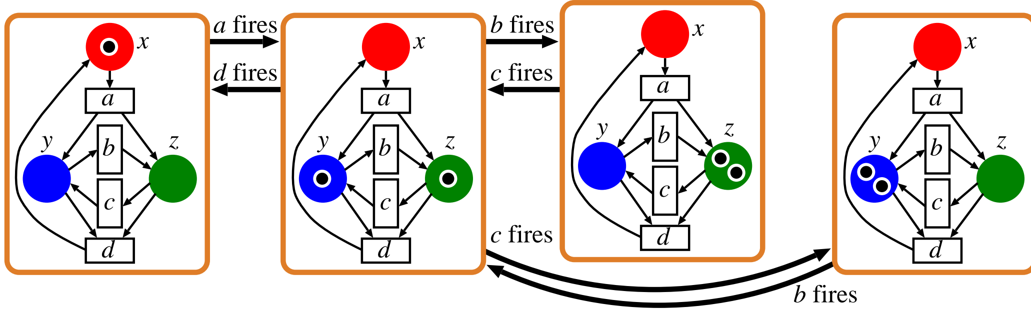


Fig. 1. The reachability graph.

and $Bot(e)$ denote the highest and lowest levels for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. A node p at level k is said to be *saturated* if it is a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, i.e., $\mathcal{B}(p) = \mathcal{B}(p) \cup \mathcal{N}_{\leq k}(\mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:Top(e) \leq k} \mathcal{N}_e$. To saturate node p once all its descendants have been saturated, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for any event e such that $Top(e) = k$. This can create new nodes at levels below k, which are saturated immediately, prior to completing the saturation of p.

If we start with the MDD that encodes the initial state $\mathbf{s}^{init}$ and saturate its nodes bottom up, the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^{init})$ at the end, because: (1) $\mathcal{N}^*(\mathbf{s}^{init}) \supseteq \mathcal{B}(r) \supseteq \{\mathbf{s}^{init}\}$, since we only add states, and only through legal event firings, and (2) $\mathcal{B}(r) \supseteq \mathcal{N}_{\leq K}(\mathcal{B}(r)) = \mathcal{N}(\mathcal{B}(r))$, since r is saturated.

In other words, saturation consists of many "lightweight" nested "local" fixed-point image computations, and is completely different from the traditional breath-first approach employing a single "heavyweight" global fixed-point image computation. Results in [6,7,8] consistently show that saturation greatly outperforms breath-first symbolic exploration by several orders of magnitude in both memory and time, making it arguably the most efficient state-space generation algorithm for globally-asynchronous locally-synchronous discrete event systems. Thus, it makes sense to attempt its parallelization, while parallelizing the less efficient breadth-first approaches would not offset the enormous speedups and memory reductions of saturation.

2.3 An example of saturation

The reachability graph of a three-place Petri net is shown in Fig. 1. Each global state is described by the local states for place x, y, and z, in that order, and we index local states by the number of tokens in the corresponding place. The reachability graph shows that three global states, (0,1,1), (0,0,2), and (0,2,0), are reachable from the initial state (1,0,0). The Kronecker description of the next-state function is shown in Fig. 2.

For instance, the matrix $\mathbf{N}_{y,b}$ of the Kronecker description indicates that firing event b will decrease the number of tokens in place y, either from 2 to 1 or from 1 to 0. Then, the saturation-based state-space generation on this model can be performed as follow (see Fig. 3).

	a	b	c	d
x	$1 \to 0$	**I**	**I**	$0 \to 1$
y	$0 \to 1$	$1 \to 0$ $2 \to 1$	$0 \to 1$ $1 \to 2$	$1 \to 0$
z	$0 \to 1$	$0 \to 1$ $1 \to 2$	$2 \to 1$ $1 \to 0$	$1 \to 0$

Fig. 2. Kronecker description of the next-state function $\mathcal{N}$.

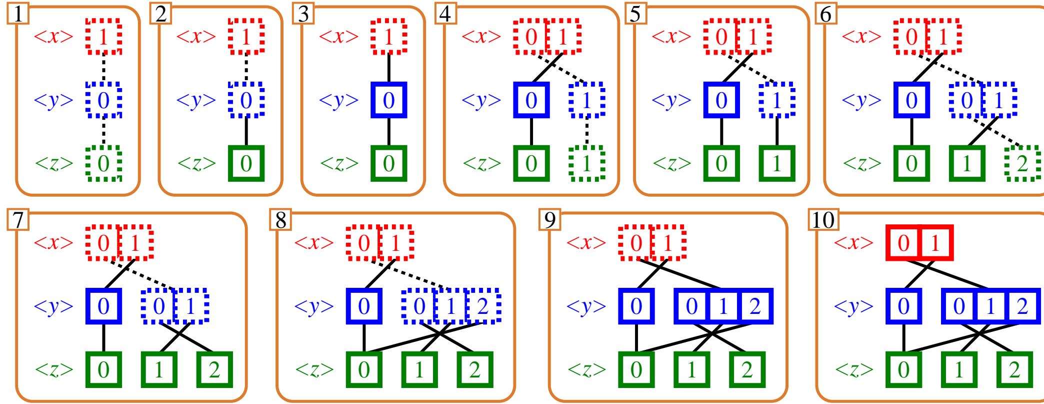


Fig. 3. Saturation example (solid nodes are saturated, dashed nodes are not).

1 **Initial configuration:** Setup the initial global state (1,0,0).

2 **Saturate node $\boxed{0}$ at level z:** No firing needs to be done, since there is no event with $Top(event) = z$. The node is saturated by definition.

3 **Saturate node $\boxed{0}$ at level y:** $Top(b) = Top(c) = y$, but neither b nor c are enabled at both levels y and z, Therefore, no firing needs to be done, and the node is thus saturated.

4 **Saturate node $\boxed{0}$ at level x:** $Top(a) = x$ and a is enabled for all levels, thus event a must be fired on the node. Since, by firing event a, local state 1 is reachable from 0 for both levels y and z, two nodes, $\boxed{1}$ at level y and node $\boxed{1}$ at level z, are created (not yet saturated), This also implies that a new global state, (0,1,1), is discovered.

5 **Saturate node $\boxed{1}$ at level z:** No firing needs to be done, since there is no event with $Top(event) = z$. Again, the node is saturated by definition.

6 **Saturate node $\boxed{1}$ at level y:** $Top(b) = y$ and b is enabled for all levels, thus event b must be fired on the node. Since, by firing event b, local state 0 is reached from 1 at level y and local state 2 is reached from 1 at level z, node $\boxed{1}$ at level y is extended to $\boxed{01}$ and node $\boxed{2}$ at level z is created. This also implies that a new global state, (0,0,2), is discovered.

7 **Saturate node $\boxed{2}$ at level z:** No firing needs to be done, since there is

no event with $Top(event) = z$. Again, the node is saturated by definition.

8 **Saturate node** $\boxed{01}$ **at level** y**:** $Top(c) = y$ and c is enabled for all levels, thus event c must be fired on the node. Since, by firing event c, local state 2 is reachable from 1 at level y and local state 0 is reachable from 1 at level z, node $\boxed{01}$ at level y is extended to $\boxed{012}$ and node $\boxed{0}$ at level z, which has been created and saturated previously, is referenced. This also implies that a new global state, (0,2,0), is discovered.

9 **Saturate node** $\boxed{012}$ **at level** y**:** Since all possible event firings have been done, the node is saturated.

10 **Saturate node** $\boxed{01}$ **at level** x**:** Since no event firing can find new global states, the root node is then saturated.

2.4 Distributed version of saturation

[4] presents *SaturationNOW*, a message-passing algorithm that distributes the state space on a NOW to study large models where a single workstation would have to rely on virtual memory. On a NOW with $W \leq K$ workstations numbered from W down to 1, each workstation w has two *neighbors*: one "below", $w-1$ (unless $w = 1$), and one "above", $w+1$ (unless $w = W$). Initially, we evenly allocate the K MDD levels to the W workstations accordingly, by assigning the ownership of levels $\lfloor w \cdot K/W \rfloor$ through $\lfloor (w-1) \cdot K/W \rfloor + 1$ to workstation w. In each workstation w, local variables $mytop_w$ and $mybot_w$ indicate the highest- and lowest-numbered levels it owns, respectively ($mytop_W = K$, $mybot_1 = 1$ and $mytop_w \geq mybot_w$ for any w). We stress that, in this distributed saturation algorithm, we use a cluster to increase the amount of available memory, not to achieve parallelism in the computation.

Each workstation w first generates the Kronecker matrices $\mathbf{N}_{k,e}$ for those events and levels where $\mathbf{N}_{k,e} \neq \mathbf{I}$ and $mytop_w \geq k \geq mybot_w$, without any synchronization. This is a simplification made possible by the fact that these matrices require little space and can be generated in isolation. Then, the sequential saturation algorithm begins, except that, when workstation $w > 1$ would normally issue a recursive call to level $mybot_w - 1$, it must instead send a request to perform this operation in workstation $w-1$ and wait for a reply. The linear organization of the workstations suffices, since each workstation only needs to communicate with its neighbors.

To cope with dynamic memory requirements, [4] uses a nested approach to reassign MDD levels, i.e., changing the $mybot_w$ and $mytop_{w-1}$ of two neighbors. Since memory load balancing requests can propagate, each workstation can effectively rely on the overall NOW memory, not just that in its neighbors, without the need for global synchronization or broadcasting. With our horizontal slicing scheme, *even an optimal static allocation of levels to workstations could still be inferior to a good, but sub-optimal, dynamic approach.* This is because, the number of nodes at a given MDD level usually increases and decreases dramatically during execution. Workstation w might be using

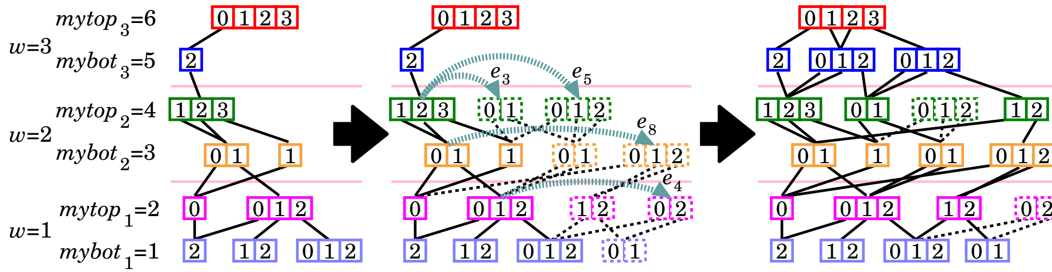


Fig. 4. Firing anticipation.

much less memory than w' at some point in time, while the reverse might occur later on. By dynamically reallocating levels between the two, such dynamic peak requirements can be better accommodated. Of course, this reallocation does not affect canonicity, since it preserves the MDD structure.

3 Speculative firing prediction

The distributed approach of [4] effectively partitions the memory load over the workstations, but it is strictly sequential. We now explore the idea of an idle workstation firing events e with $Top(e) > k$ on saturated nodes p at level k *a priori*, in the hope to reduce the time required to saturate nodes above p.

As explained in Sect. 2, an MDD node p at level k is saturated if any event e with $Top(e) = k$ has been fired exhaustively on p. However, events e with $Top(e) = l > k \geq Bot(e)$ will still need to be fired on p, if there is a path $(i_l, ..., i_{k+1})$ from a node q at level l to p, such that e is "locally enabled", i.e., $\mathcal{N}_{l,e}(i_l) \neq \emptyset, ..., \mathcal{N}_{k+1,e}(i_{k+1}) \neq \emptyset$. To accelerate the time required to saturate such hypothetical node q, our speculative prediction creates the (possibly disconnected) MDD node p' corresponding to the saturation of the result of firing e on p, and caches the result. Later on, any firing of e on p will immediately return the result p' found in the cache. Fig. 4 shows speculative firing prediction at work. In the middle, workstations 2 and 1 have predicted and computed firings for e_3 and e_5 at level 4, e_8 at level 3, and e_4 at level 2 (hence the disconnected "dashed" nodes). On the right, the nodes resulting from firing e_3 or e_8 are now connected, as they were actually needed and found in the cache: speculative prediction was effective in this case.

We stress that the MDD remains canonical, although with additional disconnected nodes. Also, even if workstation w might know a priori that event e satisfies $Top(e) > mytop_w = k \geq Bot(e) \geq mybot_w$, firing e on node p at level k can nevertheless require computation in workstation $w-1$ below, since the result p' must be saturated, causing work to propagate at levels below unless the cache can avoid it. In other words, as it is not known in advance whether the saturation of an event firing can be computed locally, consecutive idle workstations might need to perform speculative event firing together.

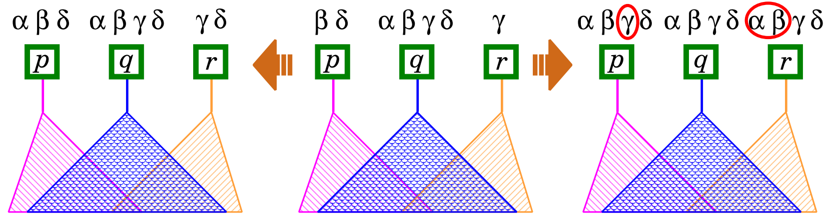


Fig. 5. History-based approach of firing prediction.

3.1 History-based approaches to speculative firing prediction

Since we do not know a priori whether event e will be fired on a node p at level k during state-space generation, the most naïve speculative firing prediction lets idle workstations exhaustively compute *all* possible firings starting "above" each node p of the MDD for $\mathcal{S}$, i.e., $\mathcal{E}_{all}(p) = \{e : Top(e) > k \geq Bot(e)\}$.

Obviously, this is effective only when $|\mathcal{E}|$ is small with respect to K, since, then, exhausting all possible firings over few events is relatively inexpensive in terms of time and space. However, for most models, this approach introduces too many nodes that never become connected to the state-space MDD.

We now motivate a more informed prediction based on firing *patterns*. For each node p at level k, let $\mathcal{E}_{patt}(p)$ be the set of events e that will be fired on p after p has been saturated, thus, $Top(e) > k$ and $\mathcal{E}_{patt}(p) \subseteq \mathcal{E}_{all}(p)$. We can then partition the nodes at level k according to their patterns, i.e., nodes p and q are in the same class if and only if $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$. Unfortunately, $\mathcal{E}_{patt}(p)$ is only known *a posteriori*, but it should be observed that most models exhibit clear firing patterns during saturation, i.e., most classes contain many nodes and most patterns contain several events.

Our goal is to *predict* the pattern of a given node p based only on the *history* of the events fired on p so far, $\mathcal{E}_{hist}(p) \subseteq \mathcal{E}_{patt}(p)$. The key idea is that, if $\emptyset \subset \mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we can speculate that the events in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ will eventually need to be fired on p as well, i.e., that $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ at the end. Fig. 5 shows an example where p, q, and r are saturated nodes at the same MDD level. The middle of Fig. 5 shows the current event firing history of these nodes at some point during runtime: $\mathcal{E}_{hist}(p) = \{\beta, \delta\}$, $\mathcal{E}_{hist}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{hist}(r) = \{\gamma\}$. The left of Fig. 5 shows the actual event firing history of these nodes after the state space is generated, i.e., their true patterns: $\mathcal{E}_{patt}(p) = \{\alpha, \beta, \delta\}$, $\mathcal{E}_{patt}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{patt}(r) = \{\gamma, \delta\}$. Applying our history-based approach, instead, will result in the firings on the right of Fig. 5: since $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ and $\mathcal{E}_{hist}(r) \subset \mathcal{E}_{hist}(q)$, the workstation owning this MDD level will fire β and γ on p and α, β, and δ on q in advance, if it is idle. Thus, the useless firings of γ on p and of α and β on q will be speculatively computed (these are highlighted with circles in Fig. 5).

Of course, we do not want to be too aggressive in our prediction. We might have that $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ for several different nodes q whose histories have few common elements in addition to $\mathcal{E}_{hist}(p)$. If, for each of these nodes q, we fire each e in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p, many of these predicted firings may be

```
FirePredict (Requests : stack, Class : array)
 while  Requests ≠ ∅  do
  (e, p) ← Pop(Requests);                    • for each event firing has been done
  Enqueue(e, E_hist(p));                     • records event firing history
  q ← Class_{p.lvl}[e];                      • the representative node for e
  if  E_hist(p) ⊃ E_hist(q)  then            • E_hist(p) is a better pattern
   Class_{p.lvl}[e] ← p;                     • p becomes the new representative
   fire e on q and cache the result;
  else if  E_hist(p) ⊂ E_hist(q)  then       • predicts event firings with the pattern
   foreach  e' ∈ E_hist(q) \ E_hist(p)  do
    fire e' on p and cache the result;
```

Fig. 6. Firing prediction algorithm.

useless, i.e., they may not be actually requested because $e \notin \mathcal{E}_{patt}(p)$. On the other hand, any prediction based on history is guaranteed to be *useful* in the rare case where the patterns of the nodes at level k are *disjoint*: i.e., if, for any two nodes p and q, either $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ or $\mathcal{E}_{patt}(p) \cap \mathcal{E}_{patt}(q) = \emptyset$.

3.2 An efficient implementation of our history-based approach

In addition to being useful, our heuristic also needs to be inexpensive in terms of memory and time overhead. Our technique, then, uses only a subset of the history and an efficient array-based method for prediction requiring $O(1)$ time per lookup and $O(K \cdot |\mathcal{E}|)$ memory overall. Each workstation w:

- stores only the c (e.g., 10) most recent elements of $\mathcal{E}_{hist}$ for its nodes.
- maintains a list $Requests_w$ containing satisfied firing request (e, p).
- uses an array $Class_k$ of size $\{e : Top(e) > k \geq Bot(e)\}$ for each level k of the MDD it owns. An element $Class_k[e]$ is a pointer to a node, initially null.

Normally, workstation w is in *saturation* mode: it computes the result of firing requests (e, p) with $Top(e) > p.lvl = k$, and records (e, p) in $Requests_w$. When w becomes idle, it turns to *prediction* mode: it removes an element (e, p) from $Requests_w$, adds e to the (truncated) history $\mathcal{E}_{hist}(p)$, and examines $Class_k(e)$. If $Class_k(e)$ = null, we set $Class_k(e)$ to p; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$, we set $Class_k(e)$ to p, and we speculatively fire the events in $\mathcal{E}_{hist}(p) \setminus \mathcal{E}_{hist}(q)$ on q; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we leave $Class_k(e)$ unchanged and we speculatively fire the events in $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p (see Fig. 6). To minimize "real work" latency, a firing request from workstation $w + 1$ switches w back to *saturation* mode, aborting any speculative firing under way.

In other words, we use $Class_k(e)$ to predict which node has the "best history" among all nodes on which e has been fired so far, and use the history of this node as our speculative firing guide for any node on which e is subsequently fired. This heuristic may suffer from "inversions": if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$ when e is fired on p, we set $Class_k(e)$ to p; later on,

further firings of q may result in $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, but $Class_k(e)$ will never be set to q, since the firing of e on q is in the cache already and will never be requested again. Nevertheless, this heuristic has minimal bookkeeping requirements, especially in saturation mode, and fast lookup times; its memory requirements are also low, since, the more workstations are idle, the faster $Requests_w$ is emptied, while $Class_k$ and the truncated history use less memory than the nodes of the MDD in practice. Sect. 4 shows that this heuristic can reduce runtime on large models. Finally, we observe that our approach can be relaxed: if we fire e on p, $Class_k(e) = q$, and $\mathcal{E}_{hist}(q) \cup \{f\} \supset \mathcal{E}_{hist}(p)$ but $f \notin \mathcal{E}_{hist}(q)$, we can still decide to speculatively fire $\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ on p; however, this aggressive approach often results in too many useless firings.

4 Experimental results

Our approach is implemented in SMARTNOW [4], the MPICH-based distributed version of our tool SMART [9]. We evaluate its performance by using saturation to generate the state space of following models.

- *Slotted ring network protocol* [22] models a protocol for local area networks where N is the number of nodes within the networks ($K = N$, $|\mathcal{S}_k| = 10$ for all k, $|\mathcal{E}| = 3N$).
- *Flexible manufacturing system* [18], models a manufacturing system with three machines to process three different types of parts where N is the number of each type of parts ($K = 19$, $|\mathcal{S}_k| = N + 1$ for all k except $|\mathcal{S}_{17}| = 4$, $|\mathcal{S}_{12}| = 3$, and $|\mathcal{S}_7| = 2$, $|\mathcal{E}| = 20$).
- *Round robin mutex protocol* [12] models the round robin version of a mutual exclusion algorithm where N is the number of processors involved ($K = N + 1$, $|\mathcal{S}_k| = 10$ for all k except $|\mathcal{S}_1| = N + 1$, $|\mathcal{E}| = 5N$).
- *Runway safety monitor* [24] models an avionics system to monitor T targets with S speeds on a $X \times Y \times Z$ runway ($K = 5(T+1)$, $|\mathcal{S}_{5+5i}| = 3$, $|\mathcal{S}_{4+5i}| = 14$, $|\mathcal{S}_{3+5i}| = 1+X(10+6(S-1))$, $|\mathcal{S}_{2+5i}| = 1+Y(10+6(S-1))$, $|\mathcal{S}_{1+5i}| = 1+Z(10+6(S-1))$, for $i = 0, ..., T$, except $|\mathcal{S}_{4+5T}| = 7$, $|\mathcal{E}| = 49 + T(56 + (Y-2)(31 + (X-2)(13+4Z)) + 3(X-2)(1+YZ) + 2X + 5Y + 3Z))$.

We run our implementation on this four models using a cluster of Pentium IV 3GHz workstations, each with 512MB RAM, connected by Gigabit Ethernet and running Red-Hat 9.0 Linux with MPI2 on TCP/IP. Table 1 shows runtimes, total memory requirements for the W workstations, and maximum memory requirements among the W workstations, for sequential SMART (SEQ) and the original SMARTNOW (DISTR), and the percentage change w.r.t. DISTR for the naïve (NAÏVE), and the history-based (HIST) speculative firing predictions; "*d*" means that dynamic memory load balancing is triggered, "*s*" means that, in addition, memory swapping occurs.

Even though the first two models have different characteristics (*slotted ring* has fixed-size nodes and numbers of levels K and events $|\mathcal{E}|$ linear in the

W	**Time** (sec)			**Total Memory** (MB)			**Max Memory** (MB)		
	DISTR	NAÏVE	HIST	DISTR	NAÏVE	HIST	DISTR	NAÏVE	HIST
Slotted ring network protocol									
$N = 200$ $\|\mathcal{S}\| = 8.38 \cdot 10^{211}$ SEQ completes in 108sec using 284MB									
2	119	−24%	−13%	286	+3%	+45%	197	+1%	+53%
4	139	−27%	−15%	286	+11%	+51%	127	+61%	+58%
8	182	−32%	−24%	286	+129%	+62%	69	+239%	+62%
$N = 300$ $\|\mathcal{S}\| = 8.38 \cdot 10^{211}$ SEQ does not complete in 5 hrs using 512MB									
2	^{s}552	^{s}+5%	^{s}−5%	962	+25%	+11%	562	+8%	+7%
4	^{d}490	> 5hrs	^{d}−16%	962	-	+34%	352	-	+12%
8	564	> 5hrs	−39%	962	-	+50%	252	-	+23%
Flexible manufacturing system									
$N = 300$ $\|\mathcal{S}\| = 3.64 \cdot 10^{27}$ SEQ completes in 55sec using 241MB									
2	79	−8%	−8%	243	+12%	+24%	121	+26%	+52%
4	91	^{d}+67	−9%	243	+102%	+30%	119	+205%	+50%
8	260	-	−30%	243	-	+42%	103	-	+47%
$N = 450$ $\|\mathcal{S}\| = 6.90 \cdot 10^{29}$ SEQ does not complete in 5 hrs using 512MB									
2	^{s}257	^{s}+12%	^{s}−14%	826	+16%	+5%	512	+15%	+7%
4	^{d}311	> 5hrs	^{d}−18%	826	-	+33%	372	-	+6%
8	959	> 5hrs	−25%	826	-	+61%	343	-	+6%
Round robin mutex protocol									
$N = 800$ $\|\mathcal{S}\| = 1.20 \cdot 10^{196}$ SEQ completes in 27sec using 290MB									
2	29	+37%	+6%	293	+110%	+85%	215	+52%	+63%
4	36	+33%	+8%	293	+348%	+109%	130	+186%	+65%
8	51	+33%	+5%	293	+807%	+148%	73	+433%	+73%
$N = 1100$ $\|\mathcal{S}\| = 3.36 \cdot 10^{334}$ SEQ does not complete in 5 hrs using 512MB									
2	^{d}65	^{s}+62%	^{s}+18%	794	+46%	+6%	379	+79%	+30%
4	47	^{s}+131%	^{d}+10%	794	+119%	+38%	265	+104%	+40%
8	56	^{d}+164%	+7%	794	+299%	+50%	173	+126%	+38%
Runway safety monitor									
$Z = 2$ $\|\mathcal{S}\| = 1.51 \cdot 10^{15}$ SEQ completes in 236sec using 314MB									
2	731	> 10hrs	−2%	332	-	+39%	191	-	+48%
4	938	> 10hrs	−8%	332	-	+88%	190	-	+30%
8	1480	> 10hrs	−22%	332	-	+128%	173	-	+13%
$Z = 3$ $\|\mathcal{S}\| = 5.07 \cdot 10^{15}$ SEQ does not complete in 10 hrs using 512MB									
2	^{s}11280	> 10hrs	^{s}−1%	962	-	+10%	595	-	+16%
4	^{d}9762	> 10hrs	^{d}−15%	962	-	+31%	371	-	+8%
8	^{d}14101	> 10hrs	^{d}−17%	962	-	+58%	359	-	+6%

Table 1
Experimental results.

parameter N; *FMS* has node size linear in N and fixed K and $|\mathcal{E}|$), both show that the pattern recognition approach improves the runtime of DISTR, more so as the number of workstations W increases, up to 39%. Indeed, NAÏVE and HIST are even faster than SEQ for *slotted ring* with $N = 200$ when $W = 2$. Furthermore, with HIST, the firing prediction is quite effective: mostly, only useful firing patterns are explored, resulting in a moderate increase in the memory requirements.

However, NAÏVE works well only if there is plenty of available memory, e.g., *slotted ring* with $N = 200$. Even then, though, increasing the number of workstations W can be counter-productive, because this increases their idle time, causing them to pursue an excess of speculative firings. This, in turn, can overwhelm the caches and the node memory and trigger expensive dynamic memory load balancing or even memory swapping, eventually slowing down the computation to levels below those of DISTR, as is the case when $N = 300$. Also, whenever W increases, the memory requirements for the most loaded workstation should decrease, as additional workstations should share the overall memory load. This holds for DISTR and HIST, but not for NAÏVE. This is even more evident for *FMS*.

Round robin mutex is a worst-case example for our approach, as no useful event firing pattern exists. We present it for fairness, but also to stress the resilience of our HIST approach. While the memory and time of NAÏVE increase dramatically because it explores many useless speculative firings, those of HIST increase only slightly, showing that HIST, being unable to help due to the lack of firing patterns, at least does not hurt much in terms of overhead.

Finally, the *RSM*, a real system being developed by National Aeronautics and Space Administration (NASA) [24], has $K = 15$, too close to W for our horizontal slicing scheme to work well. The results for SEQ are indeed much better than for any of the distributed versions, but only when SEQ can run. DISTR and HIST can still run for the second set of parameters, when SEQ fails due to excessive memory requirements. In this case, our HIST heuristic reduces the runtime with minimal additional memory overhead, confirming that event firing patterns exist in realistic models.

5 Symbolic state-space generation over a NOW

Most parallel or distributed work on symbolic state-space generation employs a vertical slicing scheme to parallelize BDD manipulations by decomposing boolean functions in breath-first fashion and distributing the computation over a NOW [14,17,26]. This allows the algorithm to overlap the image computation. However, if the slicing choice is poor, a substantial number of additional nodes is created, and it is generally agreed that finding a good slicing is not trivial [19]. Thus, some synchronization is required to minimize redundant work, and this can reduce the scalability of this approach. [13] suggests to employ a host processor to manage the job queue for load-balance purposes

Fig. 7. Vertical slicing vs. horizontal slicing with firing prediction.

and to reduce the redundancy in the image computation by slicing according to boolean functions that use an optimal choices of variables, in order to minimize the peak number of decision diagram nodes required, thus the maximum workload, among the workstations. However, no speedup is reported.

Instead, [4,23] partition the decision diagram horizontally onto a NOW, so that each workstation exclusively owns a contiguous range of decision diagram levels. Since the distributed image computation does not create any redundant work at all, synchronization is avoided. Also, with a horizontal slicing scheme. only peer-to-peer communication is required, so scalability is not an issue anymore. Yet, there is a tradeoff in that, to maintain canonicity of the distributed decision diagram, the distributed computation is sequentialized, which implies that there is no easy opportunity for speedup.

In fact, our pattern recognition approach for event firing prediction attacks this limitation while retaining the horizontal slicing scheme. However, just like the redundant work introduced by vertical slicing, our approach introduces some useless work. More precisely, even though the MDD remains canonical, additional disconnected MDD nodes can be generated. Fig. 7 shows the difference between these two approaches, where the solid boxes indicate the state space and the shaded boxes indicate the useless MDDs. Certainly, the vertical slicing approach can reorder the MDD variables to improve the node distribution, but the variable reordering operation is expensive and requires heavy synchronization. Instead, in our approach, each workstation can clean up disconnected MDD nodes at runtime without requiring any synchronization. Thus, our approach does not hurt the scalability, which is one of the advantages of a horizontal slicing scheme.

Our approach does not achieve a clear speedup with respect to the best sequential implementation. However, at least, it opens the possibility for speeding up symbolic state-space generation on a NOW in conjunction with a horizontal decision diagram slicing scheme.

6 Conclusions

We presented a pattern recognition approach to guide the speculative computation of event firings, and used it to improve the runtime of the distributed saturation algorithm for state-space generation. Experiments show that recognizing event firing patterns at runtime during saturation is effective on some

models, including that of a realistic system being developed by NASA.

We envision several possible extensions. First, while our idea is implemented for a saturation-style iteration, it is also applicable to the simpler breadth-first iteration needed in (distributed) CTL model checking. Second, having showed the potential of speculative firing prediction, we plan to explore more sophisticated, but still low-overhead, heuristics that improve the usefulness of the predicted events, while being more aggressive in the prediction when many workstation are idle. Finally, our heuristics should be augmented to include information about the current memory consumption.

7 Acknowledgment

We wish to thank Radu Siminiceanu and Christian Shelton for discussions on the anticipation approach and the referees for their helpful suggestion.

References

[1] A. Bell and B. Haverkort. Sequential and distributed model checking of Petri nets. *STTT*, 7(1):43–60, 2005.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, 1986.

[3] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[4] M.-Y. Chung and G. Ciardo. Saturation NOW. Proc. *QEST*, pp.272–281, 2004.

[5] G. Ciardo, G. Lüettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. Proc. *ICATPN*, pp.103–122, 2000.

[6] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. Proc. *TACAS*, pp.328–342, 2001.

[7] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. Proc. *TACAS*, pp.379–393, 2003.

[8] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. Proc. *CAV*, pp.40–53, 2003.

[9] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, to appear.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. 1999.

[11] S. Gai, M. Rebaudengo, and M. Sonza Reorda. A data parallel algorithm for boolean function manipulation. Proc. *FMPSC*, pp.28–36, 1995.

[12] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specification. *Formal Asp. of Comp.*, 8(5):607–616, 1996.

[13] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. Proc. *CAV*, pp.54–66, 2003.

[14] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. Proc. *CAV*, pp.20–35, 2000

[15] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. Proc. *VLSI*, pp.49–58, 1991.

[16] S. Kimura and E. Clarke. A parallel algorithm for constructing binary decision diagrams. Proc. *ICCD*, pp.220–223, 1990.

[17] K. Milvang-Jensen and A. Hu. BDDNOW : A parallel BDD package. Proc. *FMCAD*, pp.501–507, 1998.

[18] A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. Proc. *ICATPN*, pp.6–25, 1999.

[19] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability analysis using Partitioned-ROBDDs. Proc. *ICCAD*, pp.388–393, 1997.

[20] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *J. Par. and Distr. Comp.*, 47:153–167, 1997.

[21] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel implementation of BDD algorithm using a distributed shared memory. Proc. *HICSS*, pp.16–25, 1994.

[22] E. Pastor, O. Roig, J. Cortadella, and R. Badia Petri net analysis using boolean manipulation. Proc. *ICATPN*, pp.416–435, 1994.

[23] R. Ranjan, J. Snaghavi, R. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. Proc. *ICCD*, pp.356–364, 1996.

[24] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. Proc. *AVoCS*, 2004.

[25] U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. Proc. *CAV*, pp.256–267, 1997.

[26] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. Proc. *DAC*, pp.641–644, 1996.

[27] B. Yang and D. O'Hallaron. Parallel breadth-first BDD construction. Proc. *PPoPP*, pp.145–156, 1997.

DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation

Hubert Garavel[1], Radu Mateescu[2], Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier

INRIA Rhône-Alpes / VASY, 655, av. de l'Europe
F-38330 Montbonnot St Martin, France

Abstract

This paper describes Distributor, a tool for generating state spaces in a distributed manner using a set of machines connected by a network. Distributor was developed within the Cadp verification toolbox using the generic Open/Cæsar environment for on-the-fly graph exploration. It exhibits good speedups compared to sequential tools, implements on-the-fly reductions of the state space, and provides graphical features for monitoring the distributed state space generation in real time.

Key words: distributed reachability analysis, explicit state verification, labelled transition system, model checking, state space generation

1 Introduction

The verification of complex finite-state systems, whose underlying state spaces may be prohibitively large, requires an important amount of memory and computation time. A natural way of scaling up the capabilities of verification tools is by exploiting the computing resources (memory and processors) of massively parallel machines, such as clusters and grids.

We present here Distributor, a tool which constructs Labelled Transition Systems (Ltss) in a distributed manner using several machines connected by a network. Distributor implements a distributed reachability algorithm derived from [2]. Each machine is responsible for generating and storing a part of the Lts; upon termination of the distributed generation, all Lts parts generated by the machines are combined together using the Bcg_Merge tool in order to

[1] Email: Hubert.Garavel@inria.fr
[2] Email: Radu.Mateescu@inria.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

obtain the complete Lts. Additionally, Distributor can reduce the Lts on-the-fly by applying τ-compression (elimination of cycles of τ-transitions, denoting divergences of the internal behaviour of the system) or τ-confluence (a form of partial order reduction [3] preserving branching equivalence).

The current version of Distributor assumes that all machines are homogeneous in terms of their processor and operating system. As regards communication between machines, Distributor does not make strong assumptions, requiring only standard Tcp sockets and standard remote connection programs (e.g., rsh/rcp, ssh/scp, etc.) to be available. In particular, Distributor does not require the existence of a common file system (e.g., Nfs, Samba, etc.) shared between machines. Distributor runs on several platforms (Windows, MacOS, Linux, Solaris).

The machine on which Distributor is launched by the end-user is called the *local* machine, all the other ones being called *remote* machines. To perform the state space generation, Distributor will launch distributed processes, called *instances*. The list of machines and instances involved in the distributed computation must be specified in a Grid Configuration File (Gcf), described in Section 2. Typically, each instance executes on one remote machine, but there can also be several instances per remote machine, as well as some instances executing on the local machine.

The generated Lts is stored as a Partitioned Bcg Graph (Pbg), described in Section 3, which can be subsequently converted into a single Bcg file using the Bcg_Merge tool. The overall architecture [3] of Distributor and Bcg_Merge is presented in Section 4, and the graphical features for monitoring the distributed Lts generation in real time are shown in Section 5. Finally, Section 6 concludes and indicates some directions for future work.

2 Grid configuration files

A Grid Configuration File (Gcf) specifies the list of instances to be launched by Distributor, together with the various variables used for launching, connecting, and parameterizing instances on the (local) and remote machines.

Each instance corresponds to a pair (M, D), indicating that Distributor will launch a distributed process executing on machine M and storing its files in directory D (the *working directory* of the instance) located on some filesystem of M. Instances may or may not be launched on the local machine, depending on the constraints on grid usage. Indeed, clusters often distinguish

[3] This paper presents versions 3.0 of Distributor and Bcg_Merge. Versions 1.0 of these tools were developed by I. Smarandache-Sturm along the lines of [2]. Version 2.0 of Distributor was developed by A. Curic and G. Stragier, who added the graphical monitor, and version 2.0 of Bcg_Merge was developed by R. Mateescu. Versions 3.0 of Distributor (which implements a modified algorithm by C. Joubert) and Bcg_Merge were rewritten from scratch by N. Descoubes and revised by D. Bergamini and H. Garavel. The manual pages of these tools [7,6] were written by H. Garavel and R. Mateescu.

between one frontal node (i.e., the local machine) used to submit jobs to the many computing nodes (i.e., the remote machines) that perform distributed computations. Each instance is given a unique number greater or equal to 1. Numbers are assigned in the same order in which instances appear in the Gcf file. Number 0 is reserved for the process corresponding to the execution of Distributor on the local machine. Several instances may execute on the same machine, provided that their working directories are different. Also, a working directory may be either local to its machine, or shared between several machines (using Nfs, Samba, etc.).

A Gcf file can also specify the value of (predefined) variables used to control the way instances are launched and communicate with each other. A Gcf file contains a list of global directives (applicable to all instances) followed by a list of directives specific to one (or several) particular instance(s). The predefined variables correspond to the following instance parameters: the size of communication buffers (for incoming/outgoing data), the pathname of the directory in which Cadp is installed, the timeout allowed for establishing connections, the pathname of the working directory, a list of files to be copied in the working directory upon launching, the hash function used for partitioning the state space among machines, the Tcp port used for communicating with the instance, the commands used for copying files from the local to the remote machines and for launching instances on remote machines, and the user login name used for authentication when connecting to remote machines. All these variables have default values, which can be overriden by directives in the Gcf file. Also, to provide for last-minute changes, the contents of the Gcf file can be extended and/or overriden by specifying directives on the command-line upon launching Distributor.

In its simplest form, a Gcf file contains the list of remote machines to be used (a single instance will be launched on each remote machine). A more complex example of Gcf file involving all predefined variables is shown below.

```
buffer_size = 32768
cadp = /usr/local/cadp
connect_timeout = 10
directory = /home/vasy/distributor
files = graph-*.bcg
hash = 4
port = 8016
rcp = scp
rsh = ssh
user = inria
machine1.domain.org
machine2.domain.org
   user = vasy
machine3.domain.org
   directory = /users/inria/distributor
```

Note that directive "`user = vasy`" applies to both `machine1` and `machine2`. A detailed decription of the Gcf file format is available in [7].

3 Partitioned BCG graphs

Binary Coded Graphs (Bcg) is a file format used by Cadp for storing Ltss in compact, binary files. The Bcg format is equipped with a set of C libraries and tools providing a wide range of functionalities (reading and writing, exploring the transition relation, converting from/to other Lts formats, graphical displaying, etc.). A collection of Bcg files is available on-line in the Vlts benchmark suite [5], which aims at providing realistic examples of Ltss for the assessment of verification and graph manipulation tools. Note that a Bcg file provides an *explicit* representation of an Lts, containing all its states and transitions. Therefore, the Bcg format is suitable for *global* verification, which requires the Lts to be entirely constructed before verification.

The Partitioned Bcg Graph (Pbg) format implements the theoretical concept of partitioned Lts defined in [2]. A Pbg file gathers a collection of Bcg files, called "fragments" (one fragment per instance), which are stored either on the local machine (in case of a shared filesystem like Nfs, Samba, etc.), or on remote machines in the working directories associated to instances. An example of Pbg file gathering 7 fragments is shown below.

```
PBG 1.0
# PBG format by SENVA team -- http://www.inrialpes.fr/vasy/senva
# created by Distributor (C) INRIA/VASY
# (do not modify this file unless you know what you are doing)
grid: "vasy.gcf"[0]
states: partitioned
edges: incoming
initiator: 5
fragments: 7
1: states: 2667926 fragment: "fragment-1.bcg"[0] log: "1.log"[0]
2: states: 2233636 fragment: "fragment-2.bcg"[0] log: "2.log"[0]
3: states: 1919462 fragment: "fragment-3.bcg"[0] log: "3.log"[0]
4: states: 2653421 fragment: "fragment-4.bcg"[0] log: "4.log"[0]
5: states: 3326293 fragment: "fragment-5.bcg"[0] log: "5.log"[0]
6: states: 2970672 fragment: "fragment-6.bcg"[0] log: "6.log"[0]
7: states: 2666894 fragment: "fragment-7.bcg"[0] log: "7.log"[0]
```

Taken altogether, these fragments form a partition of an Lts, the states and transitions of which are distributed among the various fragments as specified in [2]. Note that, taken individually, each fragment is usually meaningless; in particular, it may be a disconnected graph, which is never the case of a state space generated from, e.g., a Lotos [4] specification. These fragments can be recombined using the Bcg_Merge tool [6], which also performs various actions, such as state renumbering.

4 Distributor and Bcg_Merge

Alternatively to the explicit Lts representation in the form of Bcg files, the Cadp toolbox also offers an *implicit* representation, implemented by the generic Open/Cæsar environment [1] for on-the-fly exploration of Ltss. Open/Cæsar specifies a language-independent Api for Ltss, which basically defines the states, labels, and transitions of the Lts, equipped with functions for comparison, hashing, accessing the initial state, and computing the successors of a given state. Open/Cæsar also provides C libraries containing a rich set of primitives for Lts exploration (transition lists, stacks, tables, etc.). This implicit representation is suitable for *local* (or on-the-fly) verification, which allows the Lts to be constructed in a demand-driven way during verification.

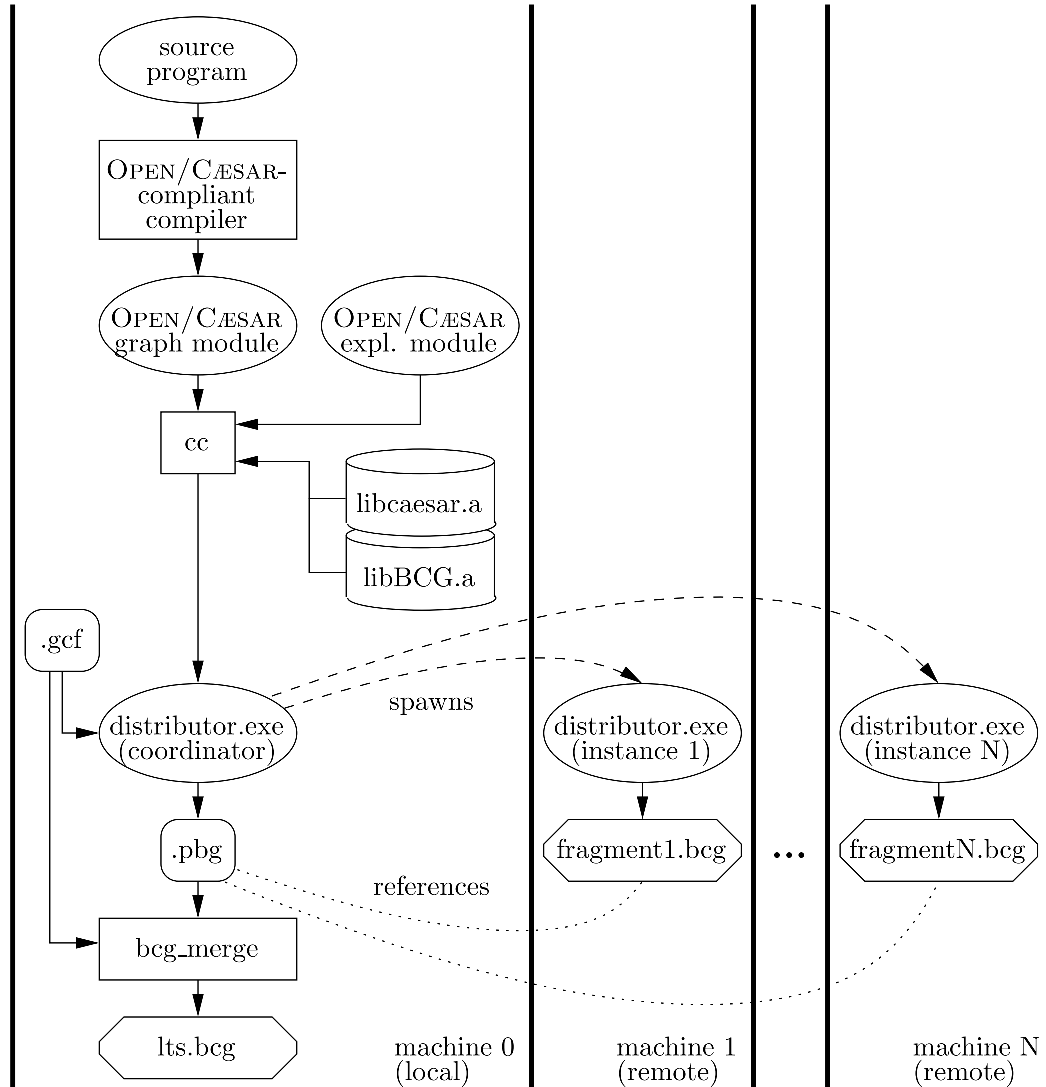


Fig. 1. Architecture of Distributor and Bcg_Merge

Distributor was developed using Open/Cæsar, which makes it available for any input language equipped with a compiler able to generate Ltss in

compliance with the Api defined by Open/Cæsar. The tool (see Fig. 1) takes as input an implicit Lts produced by an Open/Cæsar-compliant compiler and a Gcf file, and produces as output a Pbg file gathering the Lts fragments generated by the distributed execution of instances on the remote machines.

The instance numbered 0 corresponds to a *coordinator* process, which is in charge of several activities: (a) initialization of the distributed computation (parsing the Gcf file, establishing the connections from the local to the remote machines, launching the instances); (b) detection of normal termination, which occurs when all instances are idle and no messages are in transit; (c) detection of urgent termination, which may be caused either by errors during the execution of instances (e.g., failures of remote machines, memory shortages, etc.), or explicitly required by the end-user; (d) monitoring the progression of the distributed computation in real time.

After the distributed generation of the Pbg has finished, the entire Lts can be obtained as a unique Bcg file by invoking the Bcg_Merge tool.

5 Distributed graphical monitor

The distributed graphical monitor provides real time information about the distributed generation of the Pbg encoding the Lts. The monitor is driven by the coordinator, which periodically inspects the status of each instance. The monitor window is organized into five panels (see Fig. 2), each one showing a different view of the distributed computation.

The "Overview" panel in Fig. 2(a) gives, for each instance, the number of explored states (the successors of which have been computed), the number of remaining states (visited, but not explored yet), and the number of transitions in the corresponding Bcg fragment. Also, the variation of the remaining states is represented by means of a coloured box. A green (resp. orange) box indicates that the number of remaining states is increasing (resp. decreasing). A red box indicates that the instance has finished its computations; when all instances have a red box and there are no more messages in transit, the distributed exploration algorithm terminates. The "Labels" panel in Fig. 2(b) displays all different labels encountered when firing transitions during the state space generation. The "Progress" panel in Fig. 2(c) shows, for each instance, a progress bar indicating the number of states explored w.r.t. the states visited by that instance. The "Statistics" panel in Fig. 2(d) shows various global data, such as the total (and average per instance) number of visited and remaining states, of transitions, of labels, etc. Finally, the "Resources" panel in Fig. 2(e) estimates, for each instance, the corresponding memory and CPU usage.

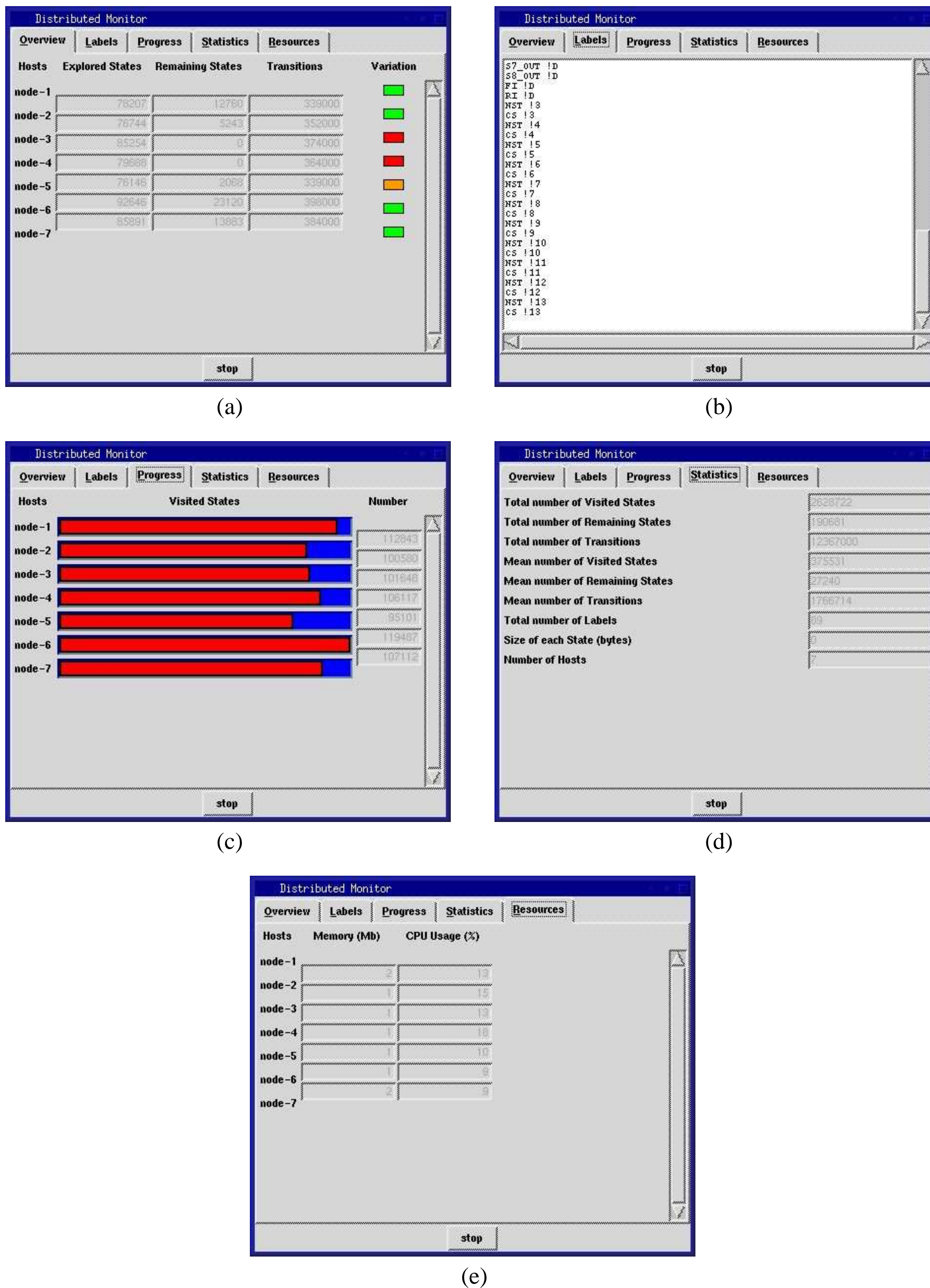


Fig. 2. Panels of the distributed graphical monitor

6 Conclusion and future work

Distributor was experimented on various examples of communication protocols and hardware devices [2], and on different computing platforms (clusters of Pcs running Linux, networks of workstations running Solaris, etc.). For

all examples, we observed quasi-linear speedups w.r.t. the Generator tool of Cadp for sequential Lts generation.

We plan to continue our work along two directions. Using Distributor, generating very large Ltss becomes easier and one is now confronted to the limits inherent to standard 32-bit machines, especially when state numbers become larger than 2^{32} and/or when Bcg files become larger than 4 Gbytes. Shifting to 64-bit machines should solve these issues and allow to overcome the current size limitations.

The Bcg_Merge tool is valuable as it allows (at least for small or medium-sized models) to verify that the Ltss generated by Distributor are identical to those generated on a single machine. For large models however, Bcg_Merge may be a bottleneck because of the aformentioned 4 Gbytes limit. For this reason, we are now considering to perform verification directly on the Pbg file itself, without invoking Bcg_Merge first. We seek to develop a Pbg_Open tool that would connect the Pbg model to the Api defined by Open/Cæsar, thus allowing the model checking and equivalence checking tools of Cadp to be applied on Pbg models directly.

Acknowledgement

We are grateful to David Champelovier and Frédéric Lang for their valuable help in porting Distributor and Bcg_Merge on the Windows operating system.

References

[1] H. Garavel. Open/Cæsar: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proc. of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, LNCS vol. 1384, pp. 68–84. Springer Verlag, March 1998. Full version available as INRIA Research Report RR-3352.

[2] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In M. B. Dwyer, editor, *Proc. of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, LNCS vol. 2057, pp. 217–234. Springer Verlag, May 2001. Revised version available as INRIA Research Report RR-4341 (December 2001).

[3] J.F. Groote and J. van de Pol. State Space Reduction using Partial τ-confluence. In M. Nielsen and B. Rovan, editors, *Proc. of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia)*, LNCS vol. 1893, pp. 383–393. Springer Verlag, August 2000. Also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000.

[4] 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 1989.

[5] Vasy. Vlts Benchmark Suite. http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html, March 2003.

[6] Vasy. Bcg_Merge Manual Page. http://www.inrialpes.fr/vasy/cadp/man/bcg_merge.html, December 2004.

[7] Vasy. Distributor Manual Page. http://www.inrialpes.fr/vasy/cadp/man/distributor.html, December 2004.

DiVinE
The Distributed Verification Environment

J. Barnat [1], L. Brim [2], I. Černá [1], P. Šimeček [2]

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

This paper presents basic concepts and the current state of a general distributed verification environment (DiVinE). The environment is meant to support the development of distributed enumerative model checking algorithms, to enable unified and credible comparison of these algorithms, and to make the distributed verification available for public use in a form of a distributed verification tool.

1 Introduction

In recent years, extensive research has been conducted in parallel and distributed model checking with the aim to push forward the frontiers of still tractable systems [13,12,11,10,4,5]. Many parallel and distributed algorithms have been developed and experimentally evaluated, mostly on a restricted set of verification problems. A few of them have been incorporated into existing verification tools. However, these distributed tools are still far from the standards met by sequential tools and in most cases their availability to the public is limited.

Another important aspect related to parallel and distributed model checking algorithms is that their performance analysis as published in original papers cannot serve for their credible comparison. This is primarily due to the fact that the hardware, the input models, and other circumstances differ from case to case making thus reported results incomparable. Most algorithms were implemented as research prototypes only using various data structures and different optimization techniques. As a consequence they simply cannot be executed on the same set of inputs. In addition, it is impossible to assure the same conditions when redoing the original experiments.

[1] Research supported by the Academy of Sciences of Czech Republic grant No. 1ET408050503

[2] Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509

In order to produce a fair comparison, the algorithms must be re-implemented on a common base and their behavior have to be re-examined on a common set of inputs and under the same conditions. This comparative evaluation can provide useful insight into their strengths and weaknesses leading to a more informed way of how to choose an appropriate algorithm for a given verification task.

In 2002 our group at the Faculty of informatics started the DiVinE project with the aim to develop a distributed LTL model checking verification tool and at the same time to provide a platform for development and comparison of distributed enumerative model checking algorithms. The main goals of DiVinE – *Distributed Verification Environment* can be summarized as follows:

(i) To use the environment as a platform for further development and experimental evaluation of enumerative parallel and distributed model checking algorithms.

(ii) To enable credible evaluation of existing enumerative algorithms with regard to their performance and characteristics under controlled conditions.

(iii) To create a ready-to-use distributed LTL model checker.

In this short presentation we aim to announce the DiVinE project to the PDMC community, introduce basic concepts and ideas used in DiVinE, describe its architecture and give the current status of work.

2 DiVinE Project

The DiVinE project splits into two main parts: DiVinE ToolSet and DiVinE Library. These parts address potential DiVinE users at two different levels: at the level of a *tool user* (DiVinE ToolSet) and at the level of a *tool developer* (DiVinE Library). The overall structure of DiVinE is depicted in Figure 1.

2.1 DiVinE ToolSet

The DiVinE ToolSet is made of a set of various model checking algorithms (referred as tools) that are accessed uniformly via a graphical user interface. An inseparable part of DiVinE ToolSet is a collection of verification problems and case studies, i.e. a collection of models and corresponding properties to be verified. This problem set should also serve for benchmark testing and evaluation. The native DiVinE specification language (DVE) has borrowed its principles from well established and by the community generally accepted description formalisms found in tools such as SPIN or UPPAAL. Each model of the system is described as a network of finite state machines with guarded transitions. The automata communicate either via shared memory (shared variables) or through buffered communication channels.

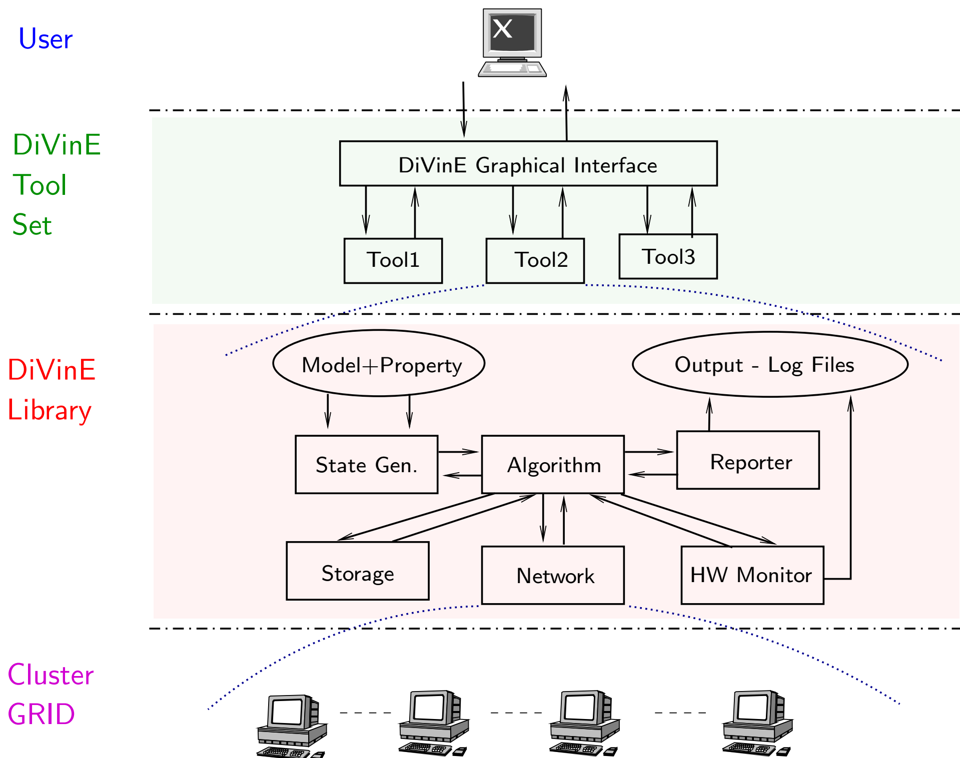


Fig. 1. DiVinE architecture.

2.2 DiVinE Library

DiVinE Library is expected to be used by the researchers who intend to design, implement and experimentally evaluate a distributed model checking algorithm. The library is therefore designed to provide the potential programmer with a plethora of functions that are typically needed for the implementation. The programmer may thus focus on the core part of the algorithm design. Furthermore, for the comparison of different algorithms it is crucial to use the same level of implementation details.

As can be seen from Figure 1, DiVinE Library is divided into several more or less independent modules.

State Generator is a module that provides functions needed for the state space generation. These include primarily the function which computes the initial state and the function which computes immediate successors of a given state. In the case of LTL model checking, the module is responsible for computing the synchronous product of system automaton and negative claim automaton, and identifying accepting states of the synchronous product. State Generator supports the programmer by additional functionality, e.g. an interface to access some structural properties of the model.

Storage module is responsible for storing states to the local memory. It provides functions for inserting states to the set of visited states, testing membership of states in the set, and removing states from this set. In addition, the module is capable of storing additional pieces of information, as the so called

appendix, for every state stored in the set. The module is able to provide a unique identifier for each stored state, hence, the algorithm can manipulate states and access their appendixes using small identifiers instead of working with large state vectors.

The purpose of *HW Monitor* and *Reporter* modules is to continuously monitor running algorithm and to feed the algorithm with information about hardware utilization as well as to produce logs describing the behavior of the algorithm during its execution. The standard POSIX signal mechanism is used to scan and log measured quantities every second.

Network module is the core module of the DiVinE Library part. This module implements basic routines for communication in the distributed setting such as send and receive primitives. In addition to the basic network primitives, the module also implements mechanisms for message buffering, functions for sending and receiving urgent messages, functions for partitioning the state space, etc. As for high-level primitives, the module mainly provides functions for synchronization. In particular, the module supports

(i) *barrier synchronization* which postpones computation on a computer until all other computers enter the barrier (this synchronization has been adopted from MPI standard),

(ii) *termination detection* which is a probe function returning `true` if all messages that have been sent have also been received and processed, and all participating computers are idle, and returning `false` otherwise. Termination detection can be repeatedly used for synchronization of participating computers as well as for collecting global numbers of visited states, sent and received messages, etc.

Other modules that are not depicted in Figure 1 include functions supporting partial order reduction, time profiling, counterexample generation, and property automaton decomposition.

3 Current State

The DiVinE project is being implemented in `C++` employing MPI standard for network communication. All the basic functions of DiVinE Library, i.e. state generator, storage, HW monitor, reporter, and network modules, have already been implemented. The current alpha version of the library is ready for public use [9].

As concerns DiVinE ToolSet, we have already implemented distributed state space generation algorithm and several distributed LTL model checking algorithms [2,3,1,6]. The algorithms were tested on models specified in native DiVinE language. DiVinE ToolSet also allows to perform guided simulation of a model and check a model for unreachable code.

Both DiVinE Library and DiVinE ToolSet have been successfully tested with `mpich`, `LAM/MPI`, and `GridMPI/YAMPII` implementations of the

MPI standard. The source codes of the DiVinE project are freely available under GNU General Public License.

4 Future Work

In the future, we would like to improve both the design and the implementation of the library, develop an appropriate user interface, and improve existing model checking algorithms as well as implement additional ones [7,8,6]. Our nearest goal however is to release a stable version of the library and DiVinE ToolSet.

Other future goals include design and implementation of functions supporting load balancing and caching of sent states. We would also like to extend the database of DiVinE models.

We have started the DivSpin project in cooperation with the Research Groups at RWTH Aachen and TU München. The goals of the project are twofold. First, to enhance DiVinE by adding support for ProMeLa specification language allowing DiVinE ToolSet to verify SPIN models. Second, to build a web-accessed platform for distributed verification and provide users with an access to appropriate hardware.

References

[1] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[2] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.

[3] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedinfs of the 3rd International Workshop on Verification and Computational Logic (VCL'02 – held at the PLI 2002 Symposium)*, pages 1–10. University of Southampton, UK, Technical Report DSSE-TR-2002-5 in DSSE, 2002.

[4] G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Proc. Workshop on Parallel and Distributed Model Checking*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[5] L. Brim and J. Barnat. Distribution of explicit-state ltl model-checking. In Thomas Arts and Wan Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier, 2003.

[6] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In *Formal Methods in Computer-Aided Design (FMCAD 2004), Austin, Texas, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2004.

[7] 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, *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FST–TCS'01)*, volume 2245 of *LNCS*, pages 96–107. Springer-Verlag, 2001.

[8] I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 49–73. Springer-Verlag, 2003.

[9] DiVinE – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[10] H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.

[11] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computertech Science*, pages 350–362. Springer-Verlag, July 2001.

[12] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th International SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1999.

[13] U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267, Berlin, Germany, 1997. Springer.

DivSPIN
A SPIN compatible distributed model checker

– Work in progress –

M. Leucker [a] M. Weber [b] V. Forejt [c] J. Barnat [c]

[a] *Institut für Informatik, Technical University of Munich, Germany*
Martin.Leucker@in.tum.de

[b] *Lehrstuhl für Informatik II, RWTH Aachen University, Germany*
michaelw@i2.informatik.rwth-aachen.de

[c] *Faculty of Informatics, Masaryk University in Brno, Czech Republic*
{xforejt,barnat}@fi.muni.cz

Abstract

This paper describes the design and implementation ideas of an extension of the parallel and distributed model checker DiVinE to a SPIN compatible distributed model checker DivSPIN. The goal of DivSPIN is to serve as user-friendly, ready-to-use system that takes up the recent theoretical and practical developments in the area of distributed model checkers and combines them with well settled operational procedures of sequential model checkers to show the benefits of parallel model checking for typical verification tasks. For this project, the research teams located at Masaryk University Brno, Czech Republic, RWTH Aachen University, and TU Munich, Germany join their efforts.

1 Introduction

SPIN is a sequential model checking tool used by thousands of people worldwide. Promela, the modeling language of SPIN, combines syntactic constructs from several popular languages, and became de facto standard specification language extensively used in sequential enumerative verification. However, when verification engineers find themselves in the situation of needing resources beyond the capabilities of a single computer, Promela models cannot be verified.

In recent years, research has been conducted in model checking algorithms which utilize the combined resources of parallel or distributed computers to further push the borders of still tractable systems. Nevertheless, most of the so-far developed algorithms have been implemented as research prototypes

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

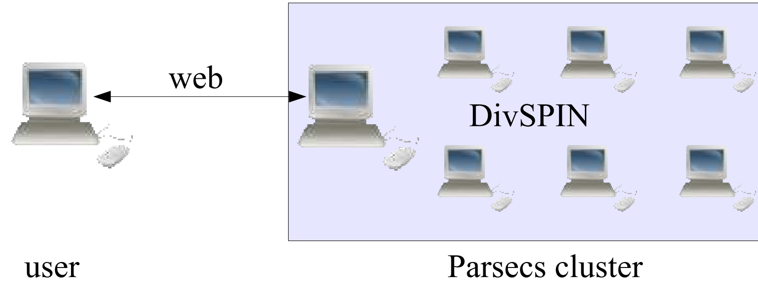


Fig. 1. Using DivSPIN

which are often not publicly available, usually undocumented, without user interface, unstable (in the sense of "prone to change"), and not optimized. These tools are mainly research vehicles, and as such not ready for widespread use by third parties.

Additionally, deployment of tools running on parallel computers is more demanding than for sequential tools. We cite high entrance costs for hardware acquisition, complex software installation procedures, but also ensuing maintenance costs. As a consequence, hardly any benchmark results of parallel and/or distributed model checking algorithms can be compared fairly, since the hardware employed for benchmarks varies from a few workstations also being used for regular tasks, to medium-sized dedicated clusters.

Recently, the DiVinE project, a distributed counterpart of standard sequential tools, has been started trying to bring advantages of distributed verification to the public. Unfortunately, DiVinE neither provides a user-friendly interface nor a widely used specification language, which are real obstacles for users that are accustomed to e. g., Promela and SPIN's graphical interface.

The goal of the DivSPIN project is to create a ready-to-use, large-scale distributed model checking tool directed at a significant part of the user base of verification tools, as well as providing hardware to run on. In particular, the goal of the project is to allow typical users of sequential model checking tools to easily access DiVinE through a user-friendly interface without the necessity of transforming Promela model into DiVinE native language.

2 Overview of DivSPIN

DivSPIN is distributed model checker running on a dedicated cluster, currently the Parsecs cluster located at RWTH Aachen University. The standard setup is shown in Figure 1. DivSPIN accepts specifications in Promela, SPIN's input language, and is able to check never claims and LTL properties. When checking LTL properties, the user can choose among several (distributed) model checking algorithms. DivSPIN is started and controlled via a web interface, to ensure that the user has a minimum of software to install and maintain.

In a typical scenario, user proceeds as follows. First, a user specifies a

model using his or her favorite editor. Furthermore, he or she formulates properties to be checked. For verification, the user selects DivSPIN's web page. Both model and property can be either uploaded from a file or typed directly into a text field. Then the user defines a verification job, henceforward called *task*. Therefore the user chooses to constrain the hardware used for the task or leave the decision to the system. Additionally, the she selects among several model checking algorithms to be used. More specifically, the user can choose several algorithms and select to stop if either the first or all have terminated. This allows comparison of different algorithms. Then the task is either executed immediately or it is submitted to a queue of waiting tasks depending on user specification and available hardware resources. During the verification task, DivSPIN reacts with status messages. Besides the web interface, a stand-alone interface, and an Eclipse plug-in are conceivable.

For the technical side, DivSPIN relies heavily on DiVinE. The DiVinE framework, described in the next section, is the back-end providing model checking algorithms. New components to be developed for DivSPIN are the Promela front-end and the web interface, both described in more detail in the next sections.

3 The Core of DivSPIN – DiVinE

The DiVinE project was started in order to support developers of *distributed* enumerative model checking algorithms, enable detailed, unified, and credible comparison of these algorithms, and make distributed verification available to the public. The project consists of two parts, the DiVinE library and the DiVinE toolset. The purpose of the library is to provide potential programmers with those necessary parts of the implementation that are common to all distributed algorithms and thus make the development of a distributed algorithm easier. The set of the algorithms that are implemented on the base of the library forms the toolset. Algorithms in the toolset have common interface, accept the same inputs, and are easily compared. This makes the DiVinE project an optimal platform for experimental evaluation of new research ideas related to parallel and distributed model checking [BBČŠ05,Div].

4 Promela Frontend

As already stated in the introduction, our choice of Promela as modeling language was driven by the wish to stay compatible to SPIN and enable SPIN's user base to change tools without effort if there is need. To interface with DiVinE, we need to obtain the meaning of Promela programs in form of state space models. Unfortunately, no complete formal semantics for Promela is available. Prior work [HN96,Wei97,Bev97] on this topic turned out to be incomplete, partly outdated, and even wrong in some places. In order to put our work on a formal basis, we decided to rigorously formalize

PROMELA, and then derive an interpreter from this specification. Generally, our main source of information were the informal description in [Hol03, ch. 7], with resorting to experimentation with SPIN to clear up ambiguities. Our requirements for PROMELA semantics are guided by pragmatism. We would like to have a simple and complete formal model, which incidentally should be effortless to implement, even with good performance in practice. Also, we need to take into account requirements imposed by our main interest, distributed model checking algorithms. For example, it must be possible to take snapshots of an interpreter's state, and restart the computation on another computer solely from this snapshot.

We have chosen a two-tier approach employing a compiler from PROMELA to a simple byte-code language, and a tiny virtual machine (VM) which interprets the compiler's output. The current implementation consists of about 3,100 lines of C code, and another 1,100 lines for testing and measuring purposes (interactive simulation, depth-/breadth-first search, hash tables).

Only the VM will be embedded into DIVSPIN, thus implementational complexity is splitted into two parts. As additional benefit, implementation of VM and compiler could be carried out in parallel, based on our formal specification, allowing rapid development. It turned out that the compiler is straight-forward to implement as well, using standard text-book techniques.

As major selling points of the VM-based approach we see the reduced complexity in semantic specification *and* implementation, and good performance results. Measurements showed that our VM generates state spaces fast enough that it will not be a dominating factor in DIVSPIN, and it is even in the same range as SPIN's state space generation. Note, that we deliberately kept actual model checking separate from the VM, as this is part of DIVINE's duties.

Generated states are almost equal in size to states generated by SPIN, with differences of a few bytes per states due to additional book keeping of our VM. A state contains all information necessary to restart the VM for further state generation, as requested above. States are represented as opaque byte sequences, thus eliminating any overhead usually imposed by converting them into a format suitable for network transmission.

5 The Web Interface

DIVSPIN's web user interface offers an easy way to initiate and control the verification task. The user interface is a web page viewable with common browsers, which remote controls verification tools running on a dedicated cluster through a server process running on its master node (cf. fig. 1). It is attractive for potential users because they avoid installing (and keeping up-to-date) any verification tools themselves, which proved to be a non-trivial task in the past due to the often prototypical nature of these tools. Besides keeping software dependencies down, users can always work with the latest and most featureful tool version without additional effort, because it is the

cluster operators' responsibility to update their software. Instead, only a web browser is needed on the client side.

The client part of the interface hides all technical details related to initiation of a distributed computation. Users can submit multiple verification jobs to the cluster, track and manipulate their status until completion, and finally retrieve results. Since it is the server part of the interface that is responsible for submitting and monitoring jobs to its associated cluster, the user can safely disconnect while his jobs are being processed. The server is capable of informing the user about the status of running jobs, for example by e-mail. If the user remains connected while verification jobs are processed, she can watch various computation statistics such as the current memory load, the number of states discovered so far, lengths of waiting queues, etc. The interface is also expected to maintain a history of the user's sessions including examined models, verified properties, and finished tasks. The user also has the possibility to set his or her personal preferences which include preferred hardware to be used, preferred algorithms to be used, etc.

The web interface will be implemented with standard web application software, using a servlet container like Apache Tomcat [TAJP]. User requests for verification tasks or their manipulation arrive at the server solely in form of HTTP requests, thus side-stepping potential problems with networks restricted by firewalls. The servlet container handles these requests and triggers corresponding actions to control the cluster. Feedback to users is provided in form of web pages. Detailed information of verification runs, like state spaces or error trails, can be downloaded as well for later use.

In order to provide a more GUI-like user experience, standard JavaScript resp. ECMAScript [ECMA99] is used to update web pages which display statistics about running and finished jobs, etc.

6 Current State

A first version of the DiVinE library [Div] is available. We have a working and stable C implementation of the virtual machine part for our Promela frontend, and first steps of interfacing it with DiVinE have already been accomplished successfully. The accompanying compiler is implemented in Java, and complete enough to compile most examples coming with the official SPIN distribution, but before production use it needs a thorough code review to assure correct implementation of our specification. Eventually, we plan to make both VM and compiler available publically.

The biggest part of remaining work needs to go into the web interface for cluster control, which is currently under development. On the hardware side, the *Parsecs* cluster at RWTH Aachen University is available for use.

References

[BBČŠ05] J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE – Distributed Verification Environment. Submitted to PDMC'05's short presentations., 2005.

[Bev97] W. Bevier. Towards an operational semantics of PROMELA in ACL2. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.

[Div] DiVinE. http://anna.fi.muni.cz/divine.

[ECMA99] European Computer Manufacturers Association. ECMA-262: ECMAScript language specification. Available at `http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM`, December 1999.

[HN96] Gerard J. Holzmann and V. Natarajan. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996.

[Hol03] Gerald J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, Boston, MA 02116, September 2003.

[TAJP] The Apache Jarkata Project. Apache Jakarta Tomcat. `http://jakarta.apache.org/tomcat/`.

[Wei97] Carsten Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd International SPIN Workshop*, 1997.

A New Reachability Algorithm for Symmetric Multi-processor Architecture

Debashis Sahoo [1]
Electrical Engineering, Stanford University, Stanford, USA

Jawahar Jain [2]
Fujitsu Laboratories of America, Inc., Sunnyvale, USA

Subramanian K. Iyer [3]
Computer Sciences, University of Texas at Austin, Austin, USA

David L. Dill [4]
Computer Science, Stanford University, Stanford, USA

Abstract

Partitioned BDD-based algorithms have been proposed in the literature to solve the memory explosion problem in BDD-based verification. A naive parallelization of such algorithms is often ineffective as they have less parallelism. In this paper we present a novel parallel reachability approach that lead to a significantly faster verification on a Symmetric Multi-Processing architecture over the existing one-thread, one-CPU approaches. We identify the issues and bottlenecks in parallelizing BDD-based reachability algorithm. We show that in most cases our algorithm achieves good speedup compared to the existing sequential approaches.

Key words: Reachability, Algorithm, Parallel, Multi-threaded

1 Introduction

A common approach to formal verification of hardware is checking invariant properties of the design. Unbounded model checking [3,9] of invariants is usually performed by doing a reachability analysis. This approach finds all the states reachable from the initial states and checks if the invariant is satisfied in these reachable states. However, exhausting the state space using the reachability approach is an intractable problem. Not surprisingly, such approaches suffer from the so-called *state explosion problem* for representing large state sets.

In practice, reachability analysis is typically done using *Reduced Ordered Binary Decision Diagrams* (OBDDs) [1,4]. A more compact representation

[1] Email: sahoo@stanford.edu
[2] Email: jawahar@fla.fujitsu.com
[3] Email: subbuk@cs.utexas.edu
[4] Email: dill@cs.stanford.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of boolean functions, *Partitioned-OBDDs* (POBDDs) [8] leads to further improvement in reachability analysis [10]. Various improvements to BDD data structures, variable ordering schemes, as well as the reachability algorithm itself have also been suggested to improve capturing the total reachable state space using reachability based verification. However, in practice the verification problem typically consumes far more resources than are typically available for even small sized problems of 100 state variables, and the gap between requirement and performance is continually growing.

The growing prevalence of, increasingly powerful, clustered high performance SMP (Symmetric Multi-Processing) machines appears to be an inevitable trend. However, it is not straightforward to devise a single algorithm to meaningfully use a very large number of processors.

Given the above two trends, it is important to develop efficient parallel verification algorithms that can appropriately exploit the SMP architecture. Though the intractability of the problem will remain, the verification time can get reduced by a significant factor.

In this paper, we show that the naive parallelization of the POBDD-based reachability analysis doesn't have good parallelism. We present a novel parallel reachability approach that improves the parallelism. Our algorithm also improves the performance of sequential POBDD based approaches drastically in some cases. This is because, in sequential POBDD-based algorithms, the relative order in which the partitions are analyzed plays a critical role in the overall performance. Finding an optimal schedule is a very hard problem. Therefore, any heuristic to find a good schedule is likely to not perform well in all cases. In a few cases, the approach can get stuck in some difficult partition and, hence, many remaining states which otherwise could have been easily computed are not reached at all. Our algorithm clearly obviates this *scheduling problem* since it runs all partitions in parallel. Also, in a parallel shared-memory environment, using our techniques of *Early Communication* and *Partial Communication*, state space traversal in some partitions can continue even while remaining partitions are proving to be difficult.

We show that in most cases our algorithm performs much better than the corresponding sequential run using 8 processors. Using our approach, we can locate error states significantly faster than other BDD based methods. We can also show that our results are much better than the standard reachability algorithms in many passing cases as well. Finally, we show that our method is more robust than the standard sequential POBDD-based reachability algorithm as it is able to solve various easy reachability instances which prove to be problematic for current POBDD approaches.

2 Preliminaries

Reachability is based on a breadth-first traversal of finite-state machines [4,9]. The algorithm takes as inputs the set of initial states and a transition relation (TR) that relates the next states a system can reach from each current state. The set of reachable states is obtained by repeatedly performing image computations until a fixed point is reached [4,9]. This is termed as the *Least Fixed Point* computation. Verification based on reachability can often be improved by the use of POBDDs [7,10,13]. Essentially, the POBDD based-reachability

algorithm performs as many steps as possible of image computation within each partition i in a step of *least fixed point* within the partition. When no more images can be thus computed, it synchronizes between partitions by considering the transitions that originate in partition i and lead out from there. The term *Communication* refers to these cross-partition image computations that are followed by transferring the computed BDDs to other partitions. Notice that the POBDD-based reachability algorithm performs a BFS which is local to individual partitions, and then synchronizes to add states that result from transitions crossing over from one partition to another. We may characterize this as a region-based BFS, where individual regions of the state space, *i.e,* the partitions, are traversed independently in a breadth first manner. We term the computation within individual partitions as a *local Least Fixed Point* computation or a local LFP computation in short.

Related Work

Several methods have been proposed to do parallel verification. Stern and Dill [16] parallelize an explicit model checker. In [17], parallelized BDDs are used for reachability analysis. Verification using parallel reachability analysis has been studied in [5,6,18]. A scalable parallel reachability analysis is presented in [6]. They perform distributed reachability using the classical BFS traversal of the state space in a parallel environment, using distributed memory. A different disjunctive partitioning approach based on iterative squaring is explored in [2]. A thread-based approach has been applied to Constraint-Based Verification in [11].

We implemented our algorithm as a multi-threaded program. We would like to compare our algorithm with other distributed approaches. However, at the time of submission of this paper, we didn't have an implementation of other distributed algorithms to compare with our approach. Therefore, we keep this as a future work.

3 Improving Parallelism in the Reachability Analysis

The reachability analysis involves construction of a TR and the actual reachability steps using the TR. We use the standard sequential approach of building the transition relation. We keep the parallelization of the construction of the transition relation as a future work. In this paper we parallelize the reachability algorithm using various heuristic improvement.

The POBDD-based algorithm given in [10] is naturally parallelizable. The local LFP computation of each partition combined with their *communication* can be processed in parallel. We have to wait for all the partitions to finish their local LFP computation and the *communication* to begin transferring the communicated states to the appropriate partition. However, empirically we find that this simple parallelization of the algorithm in [10] doesn't have much parallelism. This may be due to following reasons

High variation of BDD computations:

The performance of the image computations inside each partition depend on the BDD variable order. We call a partition an *easy partition* if the BDDs inside the partition are compact and a hard partition otherwise. For a majority of circuits, the complexity of the BDD computations can have significant variations between different partitions. In such cases, all easy partitions wait

for the hard partitions to finish their image computation, which reduces the parallelism significantly.

Depth of the local LFP computation:
Another reason for the reduced parallelism may be because the depth of the local LFP computation can vary a lot between partitions. In this case the partition with smaller depth finish faster whereas the partitions with larger depth take longer time. This results in many idle processors which reduces the parallelism.

In practice we find that a large number of partitions wait for a few hard partitions. To address this issue we use following heuristics[14] to improve the parallelism.

Early Communication: Communicate states to other partition after the least fixed point.

Partial Communication: Initiate a partial communication in an idle processor.

3.1 Early Communication

After a partition finishes its local LFP computation, we allow the partition to immediately communicate its states to the other partitions. Each partition accepts this communicated states asynchronously during their local LFP computation. This would enable the easy partitions to make progress with their subsequent local LFP computation without waiting for the hard partitions to finish. Therefore, the early communication from easy partitions to other easy partitions enables all such partitions to reach a fixed point. This is very difficult to achieve in sequential partitioned reachability analysis because such scheduling information is difficult to obtain.

If new states are *communicated* during early communication, then we restart the current image computation after adding these states. Such augmentation can make a harder image computation significantly easier in some cases. This may be because the states that would have been hard to compute in one partition can be more easily computed in another partition and then communicated to the first partition.

3.2 Partial Communication

Even after applying the above technique, we found that some partition that have completed the local LFP on their current states were waiting for other partitions to communicate some states, so that they can continue their local LFP computation. This case arises when all the easy partition finish their local LFP and need communication from a hard partition to make further progress. To improve parallelism, the active partition initiates a *communication* in an idle processor using a small subset of the state space of the hard partition. The *communication* introduces new states in the easy partitions. This enables easy partitions to make progress further with their collective least fixed point from the communicated states. Intuitively this tries to accelerate the activity among easy partitions. We found that communicating the full BDD to a different partition is very hard. Therefore, we find a small subset of state space that can be expressed with a compact BDD (High Density BDD[12]). This heuristic tries to keep all the processors busy there by improving the parallelism. Further, this heuristic can increase the number of early communication

instances. Thus, the combined effect of the partial communication and early communication improves the parallelism significantly.

```
Parallel-Reachability(n, TR, InitStates) {
    Create n partitions for InitStates
    Run in parallel for each partition i{
        After every microsteps runs
                ImproveParallelism(i) {
            Get all the communicated states
            Calculate LeastFixedPoint(Rch) in partition i
            Compute cross-over states from i to all parts
        }
    } until (No new state is found in any partition);
}
ImproveParallelism(n: Partition Number) {
    check and add all the communicated states
        if new states are added
            restart current image computation
    request a waiting partition to initiate
        partial communication procedure
}
```

Fig. 1. Parallel Reachability Algorithm

3.3 Parallel Reachability Algorithm

We present our complete parallel POBDD-based reachability algorithm as shown in Figure 1 using the techniques discussed in last section.

We run the local LFP computation combined with the *Communication* in parallel. All computation inside a partition is managed by a dedicated processor. Each processor polls for the communicated states from the other processor. After every micro-step of the image computation, each processor calls a function $ImproveParallelism$ that implements two heuristics for improving parallelism. The first heuristic is to do early communication. As a part of the first heuristic, the function checks whether other processors have communicated some states to the current partition. If it finds any processors, then it transfer all the communicated states from their corresponding partitions to the current partition. This simple check and update subroutine performed by each processor implements the early communication heuristic. The second heuristic is to do partial communication. As a part of this heuristic, every active processor checks for an idle thread. If an idle processor is found, then it gives a small subset of the state space from the current partition to the idle processor. The idle processor start a *Communication* from this subset of states to the partition associated with the idle processor.

3.4 Termination Condition

In our approach, each processor manages a partition. The processor goes back to idle state if no new states are communicated to the partition associated with that processor. One of the processor manages the global termination conditions. The processor asserts a global termination flag if all the processors are idle.

4 Engineering Issues

Our implementation of the POBDD-data structure and algorithms uses VIS-2.0 package. The VIS-2.0 package uses CUDD [15] for the BDD operations. We implemented our parallel reachability algorithm as a multi-threaded

program in a symmetric multi-processing (SMP) architecture. SMP systems can be programmed using several different methods. In a multi-threaded approach, the program divides the work across the processors by spawning multiple light-weight threads, each executing on a different processor and performing part of the calculation. Since all threads share the same program space, there is no need for any explicit communication calls. However, designing a multi-threaded FV approach using BDDs poses significant challenges.

BDD Issues in Multi-threaded Reachability: The CUDD BDD package is designed for use in a non-thread based environment. Further, there are various optimization features in CUDD, that prevent it to function correctly in a multi-threaded environment. It uses many global variables, which needs to be synchronized in a multi-threaded environment. Nevertheless, fixing this problem enable the program to behave correctly provided each thread work on their respective BDD-managers. However, this leads to a non-deterministic behavior in the BDD-computation.

The CUDD package uses various memory based optimization to boost its performance. However, such optimizations behave non-deterministically in a multi-threaded environment. Therefore, the produced computation trace is often non-reproducible and the program becomes very difficult to debug. It also results in many orders of magnitude difference in run times. Thus, the program behavior is not predictable. However, deterministic behavior of the program is very important for the evaluation of its performance. We re-engineered all the relevant features in the CUDD package that leads to a non-deterministic behavior. This enables the BDD-package to be safe to run in a multi-threaded environment and makes the program more conveniently analyzable. However, this was surprisingly painful to implement.

In addition to the above, each thread needs to synchronize based on a deterministic measure before communicating to another thread. Otherwise, the program would behave non-deterministically because of the non-determinism in the thread scheduling. We synchronize the threads using a fixed count based on the number of BDD conjunction operations and the number of sift operations during variable reordering. Further, we find that the deterministic version of the program performs as good as the non-deterministic program as described in Section 5.2.

Performance Issues on SMP machine: Further, the scheduling of the threads in an SMP machine, although improved significantly over the years, might not be optimal for our application. Each thread, in our case use separate BDD managers for carrying out various BDD operations. Therefore, if the system thread scheduler assigns the thread to a different processor, then the thread would loose all its cached data and the new processor would re-fetch all the necessary data to carry out the BDD operations. Thus, assigning a thread to a new processor would incur unnecessary large overhead. However, a very simple scheduling strategy of assigning each thread to an exclusive processor would reduce the overhead generated by the heavy cache misses significantly. On the other hand, it is quite difficult to quantify the performance penalty due the non-optimality of scheduling threads.

Performance Issues on uniprocessor machine: Furthermore, the simulated parallel execution of the multi-threaded algorithm in a uniprocessor

machine may perform better than other sequential algorithm because of the scheduling flexibility. However, the program may have large overhead due to the cache misses because of the frequent switching of threads in one processor. We find that reducing the frequency of switching of threads in a uniprocessor machine significantly improve the results. Moreover, a simulated sequential approach in an 8-CPU machine, where each thread can potentially use different processor cache improves the results further. We use explicit locks to run one thread at a time in the 8-CPU machine. We find that the performance in this simulated case is 2-6 times faster that the corresponding uniprocessor run. Thus, the uniprocessor performance is significantly penalized by the cache overhead. Therefore, we provide results from this simulated sequential approach in the 8-CPU machine in our final table to give a good overview of the parallelism achieved. However, the performance in any uniprocessor machine is much worse than the simulated sequential case in an 8-CPU machine.

5 Experimental Results

We run our experiments using default cluster size of 5000, lazy sift reordering, MLP image method on a 8-way SMP Linux machine based on Intel(R) Xeon(TM) MP CPU 2.20GHz and 8GB RAM. We run all the sequential algorithms on a Linux box with Intel(R) XEON(TM) CPU 2.20GHz and 2GB RAM. We report results only on a few VIS-verilogand industrial circuits because of limited time. Results on full VIS-verilogbenchmark circuits will appear in the final version. In keeping with the typical timeout limits set in our in-house verification tools, we set a timeout of 5000 seconds on all circuits. For sake of brevity, we present our results only on those circuits where VIS requires more than 100 seconds. Results are omitted for the circuits where all the methods timeout. We use 8 different partitions for all POBDD-based approaches. We select the partitioning variable using the method in [10]. We use same partitioning strategy for all partitioned approaches in order to perform a fair comparison.

5.1 Overview of Table

Table 1 shows our invariant check results on various public and industrial circuits. In Table 1, we separate the total reachability time into the transition relation construction time and the actual reachability time. We compare the actual reachability time taken by the following approaches: the standard approach of VIS, the simple partitioning approach and our parallel POBDD-based reachability algorithms. We compare the naive parallel approach with the successive introduction of the two heuristics for communication – early communication and partial communication. The columns in the table are arranged in the same order. The first column is the circuit name, followed by transition relation construction time, *vis*, *seq*uential POBDDs, *naive* parallelization, the parallel approach with just early communication and finally with both techniques. The final column has two parts – *8 CPUs* and *1 CPU*, which report, respectively, the total reachability time in a parallel environment using 8 CPUs and the time in a simulated sequential approach in an 8-CPU machine. The simulated sequential approach is discussed in section 4. Note that many of the sequential results are better than standard POBDD-based reachability because of the partition and communication scheduling flexibil-

ckts	TR time	vis	seq pobdd	Parallel 8 CPUs (naive)	Parallel 8 CPUs (early comm)	(early comm + partial comm) **Parallel 8 CPUs**	(early comm + partial comm) 1 CPU
(a) Industrial Circuits							
c1	36	371	T/O	T/O	T/O	**227**	286
c2	12	3346	1789	1564	93	**917**	917
c3	17	2540	T/O	T/O	T/O	**62**	228
c4	11	2236	2084	1174	161	**161**	509
(b) Few VIS-benchmark Circuits							
spprod	5	891	61	53	93	**440**	510
am2910	9	T/O	281	122	204	**356**	386
palu	3	273	4	9	8	**9**	9
s1269b-1	2	3635	T/O	T/O	59	**60**	72
s1269b-5	2	2287	T/O	T/O	55	**55**	67
blkjack-3	2	T/O	1213	470	340	**70**	98
(c) Simple Industrial Circuits							
d1	11	6	T/O	T/O	13	**13**	13
d2	15	10	11	13	45	**30**	39
d3	12	15	21	23	100	**100**	130
d4	8	11	T/O	T/O	39	**38**	60
d5	7	12	16	15	34	**37**	37

(T/O = Timeout of 5000 sec)

Table 1
Time (in sec) for Invariant Checking on a few VIS-verilog and Industrial Circuits using 8 CPUs

ity. The details of the processor utilization are presented in Section 5.3 using Gantt charts.

5.2 *Efficiency Issues*

Table 1 is composed of three different sections. Section (a) and (c), respectively shows the results on a few hard and easy industrial circuits. Section (b) shows the results on a few VIS-verilog benchmark circuits. As can be seen from the table, the resulting parallel run times with all the heuristics, i.e the last column of the table have no timeouts. They are also clearly superior to classical partitioned-reachability. The proposed parallel approach will all heuristics, is also usually superior to the less sophisticated parallel techniques. The parallel approach with only early communication, i.e the 6th column in Table 1, often works well and have less timeouts compared to the naive parallel approach. Consider the circuit *blkjack-3*, which represents the best scenario, where the results gets better and better after successive addition of the heuristics. We find that the parallel approach is usually more robust than the sequential approaches. Note that the last column shows the results of simulated sequential approach in an 8-CPU machine to demonstrate the parallelism achieved. The corresponding uniprocessor results are 2-6 times worse than the simulated sequential approach. We find that the parallelism is very small and hope to improve it in a future work.

Scheduling is a Problem Even on Easy Functions : Consider the results of some properties from an industrial design whose OBDDs are fairly small as shown in Table 1 (c). The partitioned reachability for such cases gets harder. Both the standard sequential POBDD-based reachability and naive parallel reachability falls in the trap of an inefficient computation. An early communication often helps in this case, as can be seen from the table. However, both early communication and partial communication are needed to finish all the circuits. The reachability of small circuits using 8 partitions might contribute

	redundancy [10]					
	0.3		0.5		0.7	
	Parallel	seq	Parallel	seq	Parallel	seq
c1	227	288	226	286	229	292
c2	73	386	917	917	2569	2570
c3	1492	1493	62	228	1407	T/O
c4	2967	2970	161	509	158	520
d1	26	28	13	13	92	138
d2	30	40	30	39	31	39
d3	53	67	100	130	102	133
d4	29	37	38	60	38	59
d5	13	13	37	37	37	38
s1269b-1	61	73	60	72	165	183
sp_prod	446	510	440	510	259	260

	Time in sec	
ckts	non-det	det

(a) Industrial Circuits

c1	T/O	**227**
c2	962	**917**
c3	809	**62**
c4	903	**161**

(b) Simple Industrial Circuits

d1	13	**13**
d2	24	**30**
d3	84	**100**
d4	30	**38**
d5	13	**37**

(T/O = Timeout of 5000 sec)

Table 2
Time (in sec) for Invariant Checking on the Industrial Circuits using different redundancy value in a parallel and sequential framework

Table 3
Time (in sec) for Invariant Checking on the Industrial Circuits using the non-deterministic and the deterministic program.

to some overhead in the partitioned reachability approaches.

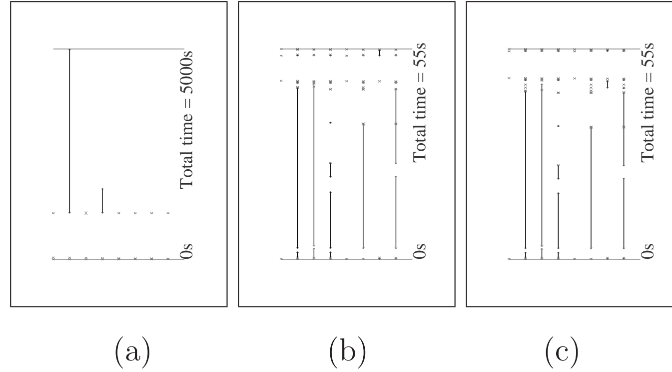
Further, we will like to comment on the relative speedup of the multi-threaded 8-CPU approach over the simulated sequential approach. This speedup is not only proportional to the algorithm but also to the choice of partitioning variables. For the same algorithm, even though the *same* partitioning variables may be provided to both the approaches, depending on the splitting choices, the amount of parallelism that is generated can vary dramatically. For example, in Table 2 it can be seen that for almost half of the entries, by varying redundancy and balancedness, the two parameters that are calculated for evaluating partitioning variables, the amount of parallelism that is generated can vary dramatically. This points to the need for an approach which can dynamically evaluate different choices in deciding the partitioning variables. Such an idea is motivated by the strong results presented in Sahoo et al. [13], where it was shown the successful BDD decisions can be taken if we generate different short traces of reachability computation for each choice and then make the required decision.

Finally, we show that the deterministic version of our program doesn't loose the performance by a great margin to the non-deterministic version. Table 3 shows the results of Invariant checking on the industrial circuits using both the non-deterministic and the deterministic version of our program. As we can see from the table, the performance of non-deterministic program is very similar to the deterministic program in the simple circuits, i.e. Table 3 (b). However, the performance of the deterministic program is better than the non-deterministic version in the hard circuits in Table 3 (a). Therefore, we strongly prefer the deterministic version to the non-deterministic version.

5.3 Improving Parallelism

Consider the reachability analysis of *s1269b-5* from the VIS Verilog benchmark suite. As shown in Table 1 (b), we perform reachability analysis using 8 partitions, each of which runs in a separate thread.

Figure 2 shows the Gantt charts of three parallel reachability analysis on *s1269b-5* circuit. We use the three charts to show the effect of the two heuristics added successively to the reachability algorithm. Figure 2(a) shows Gantt



(a) (b) (c)

Fig. 2. Parallel Reachability with successive addition of each heuristics

chart of the naive parallel reachability. Figure 2(b) shows the Gantt chart of reachability analysis when early communication is allowed. Figure 2(b) shows the Gantt chart of reachability analysis when both early communication and partial communication are allowed. Each partition is represented by a vertical broken line. The filled segment represents the *cpu time* for the partition to perform a computation. At the end of each such stage, a small cross indicates the communication of states to other partitions. A break in the line indicates that the corresponding processor is idle. However, in a multi-threaded uniprocessor environment, the processor can immediately schedule another thread for execution. The total time is the reachability time on a multi-processor machine. As we can see from the figure, more gaps are being filled with the addition of each heuristic. This shows a clear trend of improved parallelism in each case.

6 Conclusion

Partitioning based state space traversal approaches where reachability on each partition is processed independently appear very suited for parallelization. However, we find that a naive parallelization of such algorithms is often ineffective. In this paper we discuss an algorithm suitable for parallel reachability on a symmetric multi-processing architecture. We show that in most cases our algorithm achieves good speedup in a multi-processor shared memory environment, compared to the corresponding sequential run. Further, the parallel algorithm is significantly faster than both the standard sequential reachability algorithm as well as the existing partitioned approaches especially when the property is erroneous. We have made the multi-threaded program behavior deterministic. We found that the performance of both the non-deterministic and the deterministic program is similar.

Our investigation, one of the first in the area of a parallel reachability algorithm exploiting SMP architecture reveals that there are significant areas of performance improvements. These include improving scheduling of threads on various processors, selecting window functions that can potentially enhance parallelism, and communication strategies between threads to decrease number of idle CPUs.

References

[1] Bryant, R., *Graph-based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers **C-35** (1986), pp. 677–691.

[2] Cabodi, G., P. Camurati, L. Lavagno and S. Quer, *Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits.*, in: *DAC*, 1997, pp. 728–733.

[3] Clarke, E. and E. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, LNCS **131**.

[4] Coudert, O., C. Berthet and J. C. Madre, *Verification of sequential machines based on symbolic execution*, in: *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[5] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: *SPIN workshop on Model checking of software* (2001), pp. 217–234.

[6] Heyman, T., D. Geist, O. Grumberg and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, in: *CAV*, 2000.

[7] Iyer, S., D. Sahoo, C. Stangier, A. Narayan and J. Jain, *Improved symbolic Verification Using Partitioning Techniques*, in: *Proc. of CHARME 2003*, Lecture Notes in Computer Science **2860**, 2003.

[8] Jain, J., *et. al., Functional Partitioning for Verification and Related Problems*, Brown/MIT VLSI Conference (1992).

[9] McMillan, K. L., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

[10] Narayan, A., *et. al., Reachability Analysis Using Partitioned-ROBDDs*, in: *ICCAD*, 1997, pp. 388–393.

[11] Pixley, C. and J. Havlicek, *A verification synergy: Constraint-based verification*, in: *Electronic Design Processes*, 2003.

[12] Ravi, K. and F. Somenzi, *High-density reachability analysis*, in: *ICCAD*, 1995, pp. 154–158.

[13] Sahoo, D. and S. Iyer, *et. al., A Partitioning Methodology for BDD-based Verification*, in: *FMCAD*, 2004.

[14] Sahoo, D., J. Jain, S. K. Iyer, D. L. Dill and E. A. Emerson, *Multi-threaded reachability*, in: *To appear In DAC*, 2005.

[15] Somenzi, F., *CUDD: CU Decision Diagram Package ftp://vlsi.colorado.edu/pub* (2001).

[16] Stern, U. and D. L. Dill, *Parallelizing the murphy verifier*, in: *CAV*, 1997.

[17] Stornetta, T. and F. Brewer, *Implementation of an efficient parallel BDD package*, in: *DAC*, 1996, pp. 641–644.

[18] Yang, B. and D. R. O'Hallaron, *Parallel breadth-first bdd construction*, in: *symposium on Principles and practice of parallel programming* (1997), pp. 145–156.