Faculty of Informatics
Masaryk University
Czech Republic

# Platform-Dependent Verification

Habilitation Thesis
(Collection of Articles)

Jiří Barnat

October 2010

# Abstract

The computer industry is undergoing a paradigm shift. Chip manufacturers are shifting development resources away from single-core chips to a new generation of multi-core or even many-core chips. Huge clusters of multi-core workstations are easily accessible everywhere, external memory devices, such as hard disks or solid state disks, are getting more powerful both in terms of capacity and access speed. This fundamental technological shift in core computing architectures requires a fundamental change in how we ensure the quality of software. The key issue is that the verification techniques need to undergo a similarly deep technological transition to catch up with the complexity of software designed for the new hardware. It is, therefore, inevitable to come up with new techniques that allow full exploitation of the power offered by the new computer hardware to make the automated verification techniques capable of handling next-generation computer systems. In particular, this thesis demonstrates how the automated formal verification procedures, such as explicit LTL model checking or decomposition of a directed graph into strongly connected components, can be adapted to employ the computational power of clusters, multi-cored workstations, disks or graphics processing units.

The thesis is conceived as a collection of articles. The collection contains thirteen technical papers published in journals or conference proceedings, and six tool papers describing software tools released under the supervision of the author of this thesis. The author contributed to the collection mainly by formulating the ideas of results published in the articles of the collection, but also by performing numerous analyses and interpretations of experimental measurements, by writing down significant parts of texts, and by implementing parts of released software tools.

# Abstrakt

Počítačový průmysl prochází výraznou změnou výpočetního paradigmatu. Výrobci čipů se nadále nezaměřují na výrobu jednojaderných čipů, ale na výrobu vícejaderných nebo dokonce mnohojaderných čipů. I díky tomu jsou dnes běžně dostupné obrovské výpočetní klastry vícejaderných uzlů. Rostou také výkonostní parametry jako kapacita, nebo přístupová doba, všech externích paměťových médií. Tento fundamentální technologický posun v kvalitě výpočetních architektur sebou nese také posun ve způsobu, jakým je třeba zajišťovat kvalitu produkovaných výpočetních systémů. Klíčovým aspektem je zejména to, aby verifikační techniky podstoupily podobný technologický posun, a byly tak schopny zachytit komplexnost soudobých systémů. Je nezbytné vyvinout nové techniky, které umožní plně využít sílu soudobých a nadcházejících výpočetních systémů. V této habilitační práci je konkrétně demonstrováno, jakým způsobem je možné adaptovat techniky automatizované formální verifikace, jmenovitě proces ověřování modelu pro logiky lineárního času a proces dekompozice orientovaného grafu na silně souvislé komponenty tak, aby tyto techniky využily výpočetní síly klastrů, vícejaderných pracovních statnic, disků, nebo grafických karet.

Tato habilitační práce je koncipována jako soubor uveřejněných vědeckých prací (§72 odst. 3 písmena b zákona o vysokých školách). Soubor obsahuje třináct článků publikovaných v časopisech nebo konferenčních sbornících a šest článků popisujících programové nástroje, které vznikly pod supervizí autora této habilitační práce. Příspěvek autora do souboru uveřejněných prací tkví zejména ve formulaci publikovaných myšlenek, ale také v provádění nesčetných analýz a interpretací experimentálních měření, v psaní textu samotných článků a v implementaci částí zveřejněných softvérových nástrojů.

# Acknowledgments

First of all, I would like to thank to Luboš Brim for being my supervisor. I would never be where I am without his guidance, support and the courage to start the parallel model checking topic. I appreciate all the fruitful discussions, tiring squash matches, and even rare quarrels we had.

I also wish to thank all my coauthors and acknowledge all the work they did. Especially, I thank all the students participating in the development of DiVinE tool and all its spin-offs and branches.

Many thanks should also go to my wife and our daughters for their endless patience and moral support.

# Contents

## Part I

# Commentary

# Chapter 1

# Introduction

## 1.1 Motivation

The computing power of computers has increased by a factor of a million over the past couple of decades. As a matter of fact, the development effort, both in industry and in academia, has gone into developing bigger, more powerful and more complex applications. Due to various factors such as continuing miniaturization, parallel and distributed computing, etc., we may still expect a similar rate of growth in the next few decades. With the increase in complexity of computer systems, it became even more important to develop formal methods for ensuring their quality and reliability. Various techniques for automated and semi-automated analysis and verification have been designed and successfully applied to small real-life systems. However, many of these techniques are computationally demanding and memory-intensive in general and their applicability to large and complex systems routinely seen in practice these days is limited. The major hampering factor is the *state space explosion problem* due to which large industrial models cannot be efficiently handled unless more sophisticated and scalable methods are used.

A lot of attention has been paid to the development of approaches to fight the state space explosion problem [58] in the field of automated formal verification [125]. Many techniques, such as a state compaction [75], compression [94], state space reduction [124, 56, 68], symbolic state space representation [45], etc., are used to reduce the memory requirements needed to handle the verification problem with a standard sequential software tool. Employing these techniques allows user to process larger systems with the same computing power. A complementary approach suggests to employ more computational power. To that end, various verification and analysis techniques that can efficiently utilize the power of combined hardware resources have been studied. Some of the techniques are general and applicable across a broad range of computing platforms, some of them are tailored to the specific capabilities of a particular hardware architectures. Examples include techniques to fight the memory limits with an efficient utilization of external memory devices [134], techniques that introduce cluster-based algorithms to employ the aggregate power of network-interconnected computers [133, 115, 73, 7], techniques to speed-up the verification process on multi-core processors [96, 14, 113], etc. An inevitable aspect of employing combined hardware resources is parallel processing. Unfortunately, it is not the case that all the sequential solutions that are used for serial processing can be easily applied in parallel setting. On the contrary, many sequential solutions and algorithms are practically ineffective when used to utilize combined hardware resources. As a result, different solutions must have been and must be devised in order to facilitate parallel processing.

The idea of using combined resources to increase the computational power is far from being new. Attempts to use hard drives or parallel computers for verification of large systems have appeared in the very early years of the automated formal verification era. However, the inaccessibility of cheap parallel computers with sufficiently fast external memory devices together with the negative theoretical complexity results excluded these approaches from the main stream in formal verification. Moreover, thanks to the Moore's law, the performance of software tools kept improving continuously for years as the power of a single cored CPU grew. The situation changed dramatically with oncoming of multi core CPU chips. The progress in computer design over the past decades had measured several orders of magnitude with respect to various physical parameters such as power consumption, efficiency, physical size or cost. As a result, it became more efficient for chip producers to introduce multiple CPU cores on a single chip rather than to increase the speed of a single core. As the speed of a single core virtually stopped growing, every piece of software that was built upon a serial algorithm could not take the advantage of technological progress anymore. The focus of parallel and distributed-memory computing community shifted away from unique massively parallel systems competing for world records towards smaller and more cost effective systems built up from small and cheap personal computer parts. Suddenly, the need for parallel processing become rather general and wide spread in all science fields relying on complex computation operations, automated formal verification being not an exception.

Besides the parallel processing, the interest of formal verification community in specific hardware platforms has widen to graphics processing units and NVIDIA's CUDA technology, but also to contemporary external memory devices, such as solid state disks. As a matter of fact, the interest in the platform-dependent formal verification has been revived.

## 1.2 Focus of the Thesis

One particularly successful approach to automated formal verification is model checking [57, 5]. It builds upon an automated procedure that takes a model of a system and decides whether the model satisfies a given property or not. This thesis focuses, in particular, on platform dependent techniques and algorithms for model checking of formulas of *Linear Temporal Logic* (LTL) [127].

Articles included in the thesis describe results that allow implementation of scalable parallel LTL model checking algorithms. Upon the theoretical results presented in the thesis, software tools that are capable of efficient usage of aggregate computation resources of shared-memory and distributed-memory parallel architectures are presented. Thesis also describes new, the so called I/O efficient, algorithms for LTL model checking with external memory devices. Besides the LTL model checking problem, parallel algorithms for decomposing a directed graph into strongly connected components (SCCs) are described. SCC decomposition problem is inherently present in the core of many automated formal verification procedures. Finally, the platform-dependent verification of discrete nondeterministic systems is carried on to discrete probabilistic systems and systems with degradation.

## 1.3 Preliminaries

Given a model of a system, the model checking problem is to decide whether the model meets a given specification or not. For model checking purposes, the specification needs to be formalized by means of temporal logic, LTL in our case. An efficient automated procedure to decide LTL model checking problem has been introduced [138]. It employs the theory of automata over infinite words, in particular, it exploits the fact that every set of executions expressible by an LTL formula is an $\omega$-regular set and as such can be described by a *Büchi automaton*. The approach suggests to express all the system executions by a *system automaton* and all the executions violating the given LTL formula by a *property* or *negative claim automaton*. These automata are combined into their synchronous product in order to check for the existence of system executions that violate the property. The language recognized by the *product automaton* is empty if and only if no system execution is invalid.

The language emptiness problem for Büchi automata can be expressed as an *accepting cycle detection problem* in a graph. Each Büchi automaton can be naturally identified with an *automaton graph* which is a directed graph $G = (V, E, s, A)$ where $V$ is the set of vertices ($n = |V|$), $E$ is a set of edges ($m = |E|$), $s$ is an initial vertex, and $A \subseteq V$ is a set of accepting vertices. We say that a cycle in $G$ is accepting if it contains an accepting vertex. Let $\mathcal{A}$ be a Büchi automaton and $G_{\mathcal{A}}$ the corresponding automaton graph. Then $\mathcal{A}$ recognizes a nonempty language iff $G_{\mathcal{A}}$ contains an accepting cycle reachable from $s$. The LTL model-checking problem is thus reduced to the accepting cycle detection problem in the automaton graph.

Optimal sequential algorithms for accepting cycle detection use depth-first search strategy. The individual algorithms differ in their space requirements, length of the counterexample produced, and other aspects [137]. The typical algorithm used is the *Nested DFS* algorithm [61]. The idea of the algorithm is to use two interleaved depth-first searches, where the first one discovers accepting states reachable from the initial state, while the second one – the nested, checks for a self-reachability of all accepting states revealed by the first (outer) search. Several modifications of the algorithm have been suggested to remedy some of its disadvantages [76]. The well known model checker built on the Nested DFS algorithm is model checker SPIN [94, 93, 132].

The optimality of the *Nested DFS* algorithm is achieved due to the particular order in which the graph is processed. The order guarantees that no vertices of the graph are visited more than twice. In fact, all the best-known algorithms rely on the same exploring principle, namely the *postorder* as computed by the depth-first search [60]. Unfortunately, deciding the postorder is $P$-complete problem [130] and as such it is inherently sequential, which means that any algorithmic solution relying on the depth-first search postorder will have difficulties to efficiently employ contemporary parallel hardware architectures. A work-optimal scalable parallel algorithm for accepting cycle detection problem is unknown and, due to Reif [130], it is unlikely to exist.

An inseparable task of the model checking procedure is the so called *state space generation problem*. When specifying the system to be verified, the system is typically given by an initial configuration (initial state) and a function describing how the system evolves from one configuration into one or more succeeding ones. The is carried out by the so called next-state function. Such a definition of a system is referred to as an implicit definition. The state space generation problem is then a problem of enumerating all states (configurations) reachable from the initial state (initial configuration) using the next-state function. Performing the state space generation basically amounts to performing a graph traversal procedure. To guarantee termination for cyclic graphs, a graph traversal procedure keeps track of vertices (states) that have been traversed (generated). Due to the huge number of states (configurations) a system can reach, the state space generation procedure is time and memory demanding. The number of states a system can reach tends to grow exponentially with the size of the next-state function description. This is the so called *state space explosion* problem. Due to the state space explosion the amount of memory needed to store all reachable states for a real-life system typically exceeds the memory available to the algorithm, in which case the particular model checking procedure terminates incomplete. Verification approaches that are capable of detecting a violation of the verified property prior the full state space is generated are generally referred to as *on-the-fly* approaches.

# Chapter 2

# State of the Art

## 2.1 Parallel Model Checking

The need of parallel processing in automated formal verification stemmed from the desire to fight the state space explosion problem by employing aggregate memory of multiple network interconnected workstations. The crucial aspect studied at first was how to partition the state space (the set of visited states) among individual parts of the distributed-memory platform in order to take advantage of aggregate memory and parallel processing at the same time.

### 2.1.1 State Space Generation

Based on a parallel algorithm for state space generation [46] a static partitioning scheme relying on a hash function was suggested [52]. As observed by multiple researchers, the hash-based partitioning yields better space locality if only parts of the state descriptor are used as the input to the partitioning function. While there were approaches requiring the user of the tool to specify the concrete parts of the state descriptor to be used for partitioning [52, 115], other approaches employed automated or semi-automated techniques to do it [121, 122]. Techniques to load balance the set of visited states, also known as repartitioning techniques, have been suggested as well [2, 116, 111]. State space generation schemes employing probability aspects were also introduced [107, 106].

The first known public implementation of a distributed memory tool for verification of communication protocols was the parallel implementation of the Mur$\varphi$ tool [63, 133]. Active messages were used later on to improve the efficiency of the distributed-memory parallel processing with Mur$\varphi$ [141]. After the successful story of the Mur$\varphi$ tool, the distributed-memory state space generation appeared in many other verification tools, such as SPIN [115, 116], CADP [73], UPPALL [31], etc. Distributed-memory state space generation as a technique of automated formal verification also appeared in the context of Petri Nets [52, 88] and Markov chains [87, 86].

### 2.1.2 Beyond State Space Generation

The explicit model checking procedure is typically bound to linear time logic. Due to Vardi and Wolper [139], the LTL model checking problem reduces to the problem of emptiness of Büchi automata, hence to the problem of accepting cycle detection in a directed graph. Several parallel and distributed-memory algorithms for accepting cycle detection were introduced. The first implementation [17] employed the so called dependency structure to record the reachability relation among accepting states of a distributed graph and applied the topological sort algorithm [105] to detect the presence of a self-reachable accepting state. Other parallel algorithms are built upon various ideas: negative cycle detection [43, 41], property automaton decomposition [8], symbolic SCC hull detection [47], value propagation [42], or back-level edges as produced by a breadth-first search procedure [9, 10]. According to experimental evaluations, practically the best algorithm to be used for parallel accepting cycle detection combines the ideas of symbolic SCC hull detection and value propagation [15].

Besides the LTL model checking, parallel and distributed-memory algorithms for other formal verification procedures were designed. Explicit parallel and distributed-memory algorithms for verification of $\mu$-calculus [37, 38, 91] or alternation-free boolean equation systems [103] are known. Parallel explicit CTL model checking have been introduced as well [44, 40]. Techniques of state space reduction have been studied in the context of parallel processing as well. Approaches to reduce the state space modulo strong bisimulation were designed [34, 35] as well as a distributed-memory tool `LTSmin` to perform signature-based bisimulation reduction for strong and branching bisimulation [36]. Grid-enabled version of probabilistic model checker PRISM [112] has been reported too [143].

### 2.1.3 Shared-Memory Architectures

Most techniques and results known from the distributed-memory setting are straightforwardly applicable also to shared-memory architectures. However, scalability of distributed-memory solutions is often limited in shared-memory setting [12]. Therefore, shared-memory specific techniques have been developed to improve the efficiency and scalability of many parallel solutions leading in some cases almost to an optimal scalability [113]. The shared-memory specific techniques include, for example, shared communication data structures [98, 13], specific termination detection techniques [13], dual-core algorithms [96] or quite unique partitioning schemes [95].

### 2.1.4 GPU Computing

After NVIDIA's CUDA technology [62] was introduced, a lot of computational demanding task have been accelerated by GPU-aware algorithms. Examples of GPU accelerated procedures include, but are not limited to sorting procedure [77], reduce operation [85], or numerous biological and physical simulations, such as protein folding [101]. As for graph theory, successful adaptation of graph traversal algorithms were reported [83, 84] demonstrating the computational power of the CUDA device. Nevertheless, to achieve overall speedup in processing the graph to be traversed with a CUDA accelerated algorithm has to be stored in suitable data format, adjacency matrix for example.

The CUDA technology as a computing platform attracted also researches in the field of automated formal verification. The key challenge for which no satisfactory solution is known yet is how to CUDA accelerate the generation of the state space graph from the implicit definition. Preliminary attempts to do so relate to explicit model checking approach. They suggest to employ massively parallel check for enabledness of transitions emanating from the states on the frontier of the search and massively parallel execution of all the enabled transitions [66, 67].

Once the state space is generated and represented in appropriate sparse matrix like structure, many verification tasks could be accelerated using CUDA technology. This has been successfully demonstrated, for example, for explicit LTL model checking [23, 22], or verification of probabilistic systems [39].

## 2.2 Parallel Symbolic Model Checking

Symbolic approach to model checking [104] is definitely one of the most important milestones achieved in automated formal verification. The key idea of the approach is to replace the space demanding explicit enumeration of the set of states by significantly more succinct representation, and at the same time, allow for traversing of multiple edges in the state space graph at once rather than handling them one by one as done in the explicit/enumerative approach. Both goals could be achieved if the set of visited states and the next-state functions are encoded using Binary Decision Diagrams (BDDs), see e.g. [57]. The model checking procedure than reduces to manipulation of BDD structures. Unlike the explicit approach, the size of a BDD does not necessarily grow with the number of states stored in the set represented with the BDD, but rather with the irregularity of the set. For regular set of states, as produced e.g. by synchronous systems, the symbolic approach is unbeatable, but for irregular state spaces as produced typically by asynchronous systems BDDs are not that efficient.

Symbolic model checking can be adapted to parallel processing in various ways. The first option is to run a serial model checking algorithm that calls to parallel BDD manipulation routines. Such parallel BDD manipulation approaches were successfully applied to accelerate operations over large BDDs [119, 129, 135].

The second approach to adapt the symbolic model checking procedure to parallel processing mimics the state space partitioning as known from the explicit approach. To that end BDD slicing was introduced [90, 32, 89]. The set of states is a priory partitioned according to the value of BDD control variables (BDD internal nodes) and the BDD is sliced into multiple BDDs that are maintained by individual computation participating workstations. The static partitioning was found inefficient because of the network communication overhead rendered necessarily even for small verification instances. Therefore, dynamic adaptive BDD slicing were introduced later on [80]. Still the model checking process did not exhibited the expected speed-up which was, as identified later, due to the synchronous execution of individual BDD operations. This has been overcome by introducing virtually asynchronous processing over distributed BDD slices [79] that lead to up to ten-fold speedup compared to the synchronous version.

A different approach to symbolic state space generation and model checking is saturation [53, 54]. The idea of it is to avoid encoding of the transition function with a decision diagram, and thus, avoid slightly unpredictable operations over the two decision diagrams. Instead, the set of states reachable from a given set of states encoded by a BDD or an MDD (multi-valued decision diagram) is computed by direct manipulation of the internal nodes of the decision diagram representing the set of states reached so far. Unfortunately, the order in which the internal nodes of BDD or MDD are manipulated, is strictly given. The order resembles depth-first search postorder, hence, satisfactory scalable parallel technique to saturate a given BDD or MDD has not been found yet [55], some researchers even suggest to optimize the sequential algorithms rather than to parallelize them [71]. Nevertheless, horizontal partitioning [129] was employed for building up a parallel saturation procedure [49] that was improved later on with static [51] and dynamic pattern [50] for speculative execution of system transitions.

Beyond the state space generation, symbolic parallel approach to handle the verification of $\mu$-calculus formulas has been introduced as well [78].

## 2.3 Embarrassingly Parallel Model Checking

The model checking task can be viewed as one big and computation demanding procedure that is a natural candidate for being solved by means of parallel processing. The parallel solutions mentioned so far introduce multiple parallel agents that process the input data and communicate intensively to achieve the desired goal. However, this is not the only option. The whole model checking procedure can be viewed also as a bunch of many independent tasks that can be executed solely in parallel, i.e. without any communication. Such a parallel solution is generally referred to as an embarrassingly parallel approach. The difference can be nicely demonstrated on the LTL model checking problem. While the classical parallel approaches suggest to employ multiple communicating agents to detect the presence of an accepting cycle in the directed graph, the embarrassingly parallel approach suggests to take individual system executions and check every single one for its conformance with the verified property. The number of executions of a system may, however, be infinite, which renders the embarrassingly parallel approach incomplete. Therefore, the embarrassingly parallel solutions could rather be viewed as fast bug finding techniques. Examples of embarrassingly parallel approaches include parallel randomized state space search [64] or parallel guided counter-example generation [131].

Regarding the LTL model checking procedure, the order in which the vertices of the product automaton graph are explored plays significant role provided the graph contains an error state or accepting cycle to be discovered. With good traverse order the discovery of an error is a matter of relatively small number of steps of the underlying algorithm. An embarrassingly parallel approach to LTL model checking that instanciates multiple standard sequential procedures in parallel each with a randomly modified order of exploration has been introduced [97].

## 2.4 SCC Decomposition

The problem of decomposition of a directed graph into its strongly connected components is a fundamental graph problem inherently present in many scientific and commercial applications. The problem is defined as follows. Let $G$ be a directed graph, i.e. $G$ is a pair ($V$, $E$), where $V$ is a set of vertices, and $E \subseteq V \times V$ is a set of edges. Let $E^*$ be a transitive and reflexive closure of $E$ and $s, t \in V$ two vertices. We say that vertex $t$ *is reachable* from vertex $s$ if $(s, t) \in E^*$. A set of vertices $C \subseteq V$ is *strongly connected*, if for any vertices $u, v \in C$, we have that $v$ is reachable from $u$. A *strongly connected component* (SCC) is a *maximal* strongly connected $C \subseteq V$, i.e. such that no $C'$ with $C \subsetneq C' \subseteq V$ is strongly connected. The problem of SCC decomposition is the problem of identification of all strongly connected components for a given graph.

As for the automated formal verification, the SCC decomposition problem is used as a subroutine in many algorithmic solutions. For example, the SCC decomposition algorithm is employed for verification of probabilistic systems, state space reduction by $\tau$-confluence, verification of systems with fairness constraints, or verification of liner time properties given by other than Büchi automata. SCC-based algorithms can also be used directly for LTL model checking. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counterexamples in general [69].

An efficient algorithmic solution to this problem is due to Tarjan [136], who showed that, given a graph with $n$ vertices and $m$ edges, it is possible to identify and list all strongly connected components of the graph in $O(n+m)$ time and $O(n)$ space. Unfortunately, the Tarjan's solution builds upon the depth-first search postorder and as such it is limited to sequential computing paradigms, hence inappropriate for contemporary parallel computing platforms. The existence of an work-optimal scalable parallel algorithm for SCC decomposition is an open problem. All the so far known parallel solutions to the problem exhibit unoptimal time complexity.

Different approaches suitable for parallel processing have been considered. See e.g. [74, 59, 3] for algorithm that works in $O(log^2 n)$ time, but requires $O(n^{2.376})$ parallel processors, or [142] for randomized parallel algorithm for the problem. Another parallel algorithm for SCC decomposition exploits the fact that it is possible to efficiently compute in parallel the set of vertices reachable from a certain vertex or set of vertices [72]. The general idea of the algorithm is to repeatedly pick a vertex of the graph and identify the component to which it belongs, by using the forward and a backward parallel reachability procedures. The algorithm proved to be efficient enough in practice, which resulted in several theoretical improvements of it [123, 117]. The worst time complexity of the algorithm is $O(n \cdot (n + m))$. Nevertheless, the algorithm exhibits $O(m \cdot log\, n)$ expected time [72]. A completely different strategy to detect SCC in parallel was introduced in [123]. The algorithm employs value forward value propagation to partition the graph into subgraphs respecting the SCCs. Each subgraph as computed by the algorithm is rooted, hence subsequent backward reachability identifies exactly the leading component of the subgraph. The algorithm performs well for graphs with many small components, however, for graphs with large components it is easily outperformed by other parallel algorithms.

## 2.5 Model Checking with Disks

Efficient usage of memory hierarchies is an established research topic [118]. Specialized algorithms were devised to efficiently utilize external-memory block devices. The efficiency is of such the algorithms is typically measured using the so called I/O (input/output) complexity [1]. First of all, general graph traversal algorithms (state space generation, in the context of formal verification) were adapted to become I/O efficient. To that end the delayed duplicate detection was introduced [48] and further improved [144, 6, 81] or specialized for undirected graphs [108, 109].

Employing disk to fight the state explosion problem in formal verification has started by the disk extension of the verification tool Mur$\varphi$ [134, 126]. The external devices were also used to reconstruct the counterexample when applying the sweep-line heuristics search [110].

As for problems beyond the state space generation. First results published employ a generic reduction of model checking problem to the reachability problem [33]. Unfortunately, such a reduction resulted in a quadratic grow in the space demands, which effectively eliminated the possibility of complete search. There were heuristics used instead trying to prove the existence of a counterexample. We have seen random walks strategy [102], or iterative deepening and $A^*$ algorithms to be used [99, 100]. Another incomplete model checking approach suggested builds on the fact that new transitions tend to lead to new states or to a states in recent breadth-first search levels [114].

The quadratic space overhead in the I/O efficient LTL model checking was avoided later on [24] and further improved by introducing the so called merge omissions [26] that allowed for more efficient delayed duplicate detection in the later stages of the computation. Various formulas for actual omissions were introduced [70]. A completely different technique for trading time for space has been suggested and is now referred to as the semi-external approach to LTL model checking problem [65].

A problem related to I/O efficient verification, delayed duplicate detection in particular, exists and is known as the streaming state space problem [92].

# Chapter 3

# Thesis Contribution

This habilitation thesis is conceived as a collection of articles. Summary of results achieved is given in four sections. Each section groups together results with a common research topic and lists the concrete percentage of contribution by the author of this thesis for each relevant article in the collection. An extra section is then devoted to the related software tools that were solely supervised and partly developed by the author of this thesis.

## 3.1 Parallel and Distributed-Memory LTL Model Checking

### Achieved results

**Distributed-Memory LTL Model Checking** Parallel LTL model checker DiVinE [18] has been successfully adapted to various contemporary hardware platforms. Initially the tool was intended to aggregate computational power and system memory of multiple network interconnected workstations (clusters) in order to facilitate the verification of large model checking instances [17, 7]. We demonstrated that the tool succeeded the mission in terms of both the speedup achieved due to parallel processing and the ability of processing large model checking problem instances [140].

**Shared-Memory LTL Model Checking** In the light of technological shift towards shared-memory systems, we described relative advantages and disadvantages of shared versus private hash tables [29]. These were evaluated, both theoretically and practically, in a prototype implementation [14]. Later we have further improved the scalability of the tool and were able to demonstrate that the parallel processing even with an unoptimal algorithm outperforms highly efficient work-optimal sequential model checker SPIN [12].

**On-the-fly Parallel Algorithm for LTL Model Checking** Though, the optimality of the algorithm employed for parallel processing is an issue. There is an important subclass of LTL for which optimal scalable parallel algorithm exists [47]. However, this algorithm suffers from not being an on-the-fly algorithm. Since the on-the-fly verification is an important practical aspect, we have devised a modification of this algorithm that allows for on-the-fly verification in most verification instances [15].

**CUDA Accelerated LTL Model Checking** Finally, recent technological advancements in GPU computing made available a new rather specific computing platform – the NVIDIA's CUDA technology [62]. It allows for acceleration of computation intensive applications with GPU hardware. We have succeeded to adapt algorithms for accepting cycle detection to CUDA framework and demonstrated significant speedup of the LTL model checking process with CUDA technology [23].

### Articles in Collection

[29] **J. Barnat** and P. Ročkai. Shared Hash Tables in Parallel Model Checking. *ENTCS*, 198(1):79–91, 2008.

Author's contribution: 50%, significant part of the writing, main idea.

[12] **J. Barnat**, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):139–153, 2010.

Author's contribution: 33%, significant part of the writing, analysis of experimental results and formulation of conclusions.

[140] K. Verstoep, H. Bal, **J. Barnat**, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

Author's contribution: 25%, DiVinE architecture consultant, marginal part of writing.

[15] **J. Barnat**, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

Author's contribution: 70%, most of the writing, main idea, implementation, and experimental validation.

[23] **J. Barnat**, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE Computer Society, 2009.

Author's contribution: 25%, most of the writing, main ideas.

## 3.2 I/O Efficient Verification

### Achieved results

**I/O Efficient LTL Model Checking** Due to the state space explosion problem, the graph to be searched for the presence of an accepting cycle tends to be extremely large. For that reason the LTL model checking verification procedure suffers from limited applicability w.r.t the size of model checking instance if performed on a single workstation. Reduction techniques [57, 5] are simply not strong enough to solve the problem. To move the frontier of still tractable systems a little bit further external memory devices (disks) are an option. We were first to show that the LTL model checking process can be done I/O efficiently with the same space complexity as the standard pure in memory solution [24].

**Improved Delayed Duplicate Detection Technique** The idea of LTL model checking with external memory devices is to keep the track of vertices that have been explored by the algorithm on the external memory. Unfortunately, in order to access the external memory efficiently, the standard work-flow of a graph traversal algorithm has to be modified. This modification is referred to as the *delayed duplicate detection* [108, 109, 120, 134]. According to our experimental measurements, the standard delayed duplicate detection technique becomes rather ineffective once the graph traversal procedure is about to complete the search. We have, therefore, defined an improved version of the work-flow and demonstrated its positive impact on I/O efficient verification [26]. Unfortunately, not all the parallel graph traversal algorithms that are suitable for in memory computing are compatible with our new work-flow modification. Hence, we have also defined a criterion for deciding the compliance of a graph traversal algorithm with our modification – the so called *revisiting resistance*.

**Parallel I/O Efficient Model Checking** We have also investigated how parallel disks can be combined to further improve the I/O efficient LTL model checking procedure [25] and whether the recent introduction of flash memory disks have some implications on the field of I/O efficient processing [11].

### Articles in Collection

[24] **J. Barnat**, L. Brim, and P. Šimeček. I/O Efficient Accepting Cycle Detection. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.

Author's contribution: 33%, analyses of experimental results, significant part of writing.

[11] **J. Barnat**, L. Brim, S. Edelkamp, D. Sulewski, and P. Šimeček. Can Flash Memory Help in Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 150–165. Springer-Verlag, 2008.

Author's contribution: 25%, analyses of experimental results, formulation of conclusions, significant part of writing.

[26] **J. Barnat**, L. Brim, P. Šimeček, and M. Weber. Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).*, volume 4963 of *LNCS*, pages 48–62. Springer, 2008.

Author's contribution: 25%, revisiting resistant work-flow identification, analyses of experimental results, formulation of conclusions, significant part of writing.

[25] **J. Barnat**, L. Brim, and P. Šimeček. Cluster-Based I/O Efficient LTL Model Checking. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 635–639. IEEE Computer Society, 2009.

Author's contribution: 33%, analyses of experimental results, formulation of conclusions, significant part of writing.

## 3.3 SCC Decomposition

### Achieved results

**OBF Graph Decomposition Procedure** We have developed a new parallel technique to partition a directed graph into multiple SCC respecting parts – the so called *OBF* technique [28]. The technique is unique as it can partition the graph into a number of subgraphs in linear time. A such it combines the good properties of the forward-backward strategy [72] that works in linear time but produces only a constant number of subgraphs, and the value propagation approach [123] that identifies a number of subgraphs, but requires quadratic time.

**Recursive OBF Algorithm for Parallel SCC Decomposition** The OBF technique has been further improved and used recursively to build a new standalone parallel algorithm for SCC decomposition – *Recursive OBF* [27]. According to our experimental evaluation over various types of directed graphs, the new algorithm outperforms all the known parallel SCC decomposition algorithms known so far.

### Articles in Collection

[28] **Jiří Barnat** and Pavel Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

Author's contribution: 60%, OBF technique, complete writing.

[27] **J. Barnat**, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation Advance Access*, 2010.

Author's contribution: 50%, Recursive OBF algorithm idea.

## 3.4 Verification of Probabilistic Systems

### Achieved results

**Parallel Quantitative LTL Model Checking** Quantitative analysis of probabilistic systems has been studied mainly from the global model checking point of view. In the global model checking problem, the goal of the verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in the local model checking approach the probability of satisfaction is computed only for the set of initial states. We devised parallel local model checking procedure and demonstrated that with the platform dependent local model checking procedure we were able to reduce the runtime needed for verification from days to minutes [20].

**Degradation Concept** The quantitative model checking procedure was extended also to the systems with *degradation* [30]. Under some condition, systems with degradation may be viewed as the standard probabilistic systems – Markov Decision Processes (MDP's) [128]. Rather theoretical result we obtained was that the degradation properties can distinguish probabilistic systems (MDP's) that are indistinguishable by means of the standard probabilistic logics such as LTL, PCTL [82] or PCTL$^*$ [4].

### Articles in Collection

[20] **J. Barnat**, L. Brim, I. Černá, M. Češka, and J. Tůmová. Local Quantitative LTL Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 53–68. Springer-Verlag, 2008.

Author's contribution: 20%, analyses of experimental results, algorithmics, formulation of conclusions, significant part of writing.

[30] **J. Barnat**, I. Černá, and J. Tůmová. Quantitative Model Checking of Systems with Degradation. In *Proceeding of the Sixth International Conference on Quantitative Evaluation of Systems (QEST 2009)*, pages 21–30. IEEE, 2009.

Author's contribution: 33%, concept of degradation, relation to probabilistic systems, writing.

## 3.5 Tools and Tool Papers

In this section we describe software tools that were solely supervised and partly developed by the author of this thesis.

**DiVinE, DiVinE Cluster** LTL model checker built over the MPI standard allowing efficient utilization of computational resources of a cluster of workstations (*Obsolete*).

**DiVinE-MC** Clone of DiVinE dedicated for usage solely on multi-cored CPUs with shared memory architecture (*Obsolete*).

**DiVinE-CUDA** Clone of DiVinE dedicated for usage with NVIDIA's CUDA technology on workstations with appropriate graphics processing units.

**DiVinE 2.x** New implementation of parallel LTL model checker with the combined capabilities of previous DiVinE versions. With the release of DiVinE 2.x tool DiVinE and DiVinE-MC became obsolete.

**ProbDiVinE** Tool for qualitative model checking of probabilistic systems capable of employing aggregate computational power of a cluster of workstations.

**ProbDiVinE-MC** Tool for quantitative model checking of probabilistic systems capable of efficient utilization of multiple cores on a shared-memory platform.

### Articles in Collection

[18] **J. Barnat**, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

Author.'s contribution: 40%, Overall concept of the tool, implementation of multiple parallel accepting cycle detection algorithms.

[14] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

Author's contribution: 33%, LTL model checking algorithmics.

[22] J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. *Electronic Proceedings in Theoretical Computer Science (PDMC 2009)*, 14:107–111, 2009.

Author's contribution: 40%, algorithmics, algorithm engineering.

[16] Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.

Author's contribution: 33%, tool road-map, processing of precompiled models.

[19] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 215–216. IEEE Computer Society, 2007.

Author's contribution: 20%, algorithmics, tool distribution.

[21] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Probdivine-mc: Multi-core ltl model checker for probabilistic systems. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 77–78, Washington, DC, USA, 2008. IEEE Computer Society.

Author's contribution: 20%, algorithmics.

# Bibliography

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In G. Bilardi, A. G. Ferreira, R. Lüling, and J. D. P. Rolim, editors, *Proceeding of the Parallel Computing Conference PARCO'97 (Bonn, Germany)*, volume 1253 of *LNCS*, pages 207–218. Springer, 1997.

[3] N. Amato. Improved Processor Bounds for Parallel Algorithms for Weighted Directed Graphs. *Information Processing Letters*, 45(3):147–152, 1993.

[4] Adnan Aziz, Vigyan Singhal, Felice Balarin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 155–165, London, UK, 1995. Springer-Verlag.

[5] Ch. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] Tonglaga Bao and Michael Jones. Time-efficient model checking with magnetic disk. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 526–540. Springer, 2005.

[7] J. Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Masaryk University Brno, Faculty of Informatics, 2004.

[8] J. Barnat, L. Brim, and I. Černá. Property driven distribution of Nested DFS. In *Proc. Workshop on Verification and Computational Logic*, number DSSE-TR-2002-5 in DSSE Technical Report, pages 1–10. University of Southampton, UK, 2002.

[9] 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.

[10] J. Barnat, L. Brim, and J. Chaloupka. From Distributed Memory Cycle Detection to Parallel LTL Model Checking. *Electronic Notes in Theoretical Computer Science*, 133(1):21–39, 2005.

[11] J. Barnat, L. Brim, S. Edelkamp, D. Sulewski, and P. Šimeček. Can Flash Memory Help in Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 150–165. Springer-Verlag, 2008.

[12] J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):139–153, 2010.

[13] J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core ltl model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[14] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

[15] J. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

[16] J. Barnat, L. Brim, and P. Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.

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

[18] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[19] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 215–216. IEEE Computer Society, 2007.

[20] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Local Quantitative LTL Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 53–68. Springer-Verlag, 2008.

[21] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE-MC: Multi-core LTL Model Checker for Probabilistic Systems. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 77–78, Washington, DC, USA, 2008. IEEE Computer Society.

[22] J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. *Electronic Proceedings in Theoretical Computer Science (PDMC 2009)*, 14:107–111, 2009.

[23] J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE Computer Society, 2009.

[24] J. Barnat, L. Brim, and P. Šimeček. I/O Efficient Accepting Cycle Detection. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.

[25] J. Barnat, L. Brim, and P. Šimeček. Cluster-Based I/O Efficient LTL Model Checking. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 635–639. IEEE Computer Society, 2009.

[26] J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).*, volume 4963 of *LNCS*, pages 48–62. Springer, 2008.

[27] J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *To appear in Journal of Logic and Computation*, 2010.

[28] J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

[29] J. Barnat and P. Ročkai. Shared Hash Tables in Parallel Model Checking. *ENTCS*, 198(1):79–91, 2008.

[30] J. Barnat, I. Černá, and J. Tůmová. Quantitative Model Checking of Systems with Degradation. In *Proceeding of the Sixth International Conference on Quantitative Evaluation of Systems (QEST 2009)*, pages 21–30. IEEE, 2009.

[31] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proc. 12th Conference on Computer-Aided Verification CAV00*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

[32] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-fly Symbolic Model Checking. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proc. 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00), Austin, Texas*, volume 1954 of *LNCS*, pages 390–404, 2000.

[33] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Computer Science*, 66(2), 2002.

[34] Stefan Blom and Simona Orzan. Distributed State Space Minimization. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[35] Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005.

[36] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and Symbolic Reachability. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.

[37] B. Bollig, M. Leucker, and M Weber. Parallel model checking for the alternation free mu-calculus. In T. Margaria and W. Yi, editors, *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[38] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*. Springer-Verlag Inc., 2002.

[39] D. Bosnacki, S. Edelkamp, and D. Sulewski. Efficient Probabilistic Model Checking on General Purpose Graphics Processors. In *Model Checking Software (SPIN 2009)*, volume 5578 of *LNCS*, pages 32–49. Springer, 2009.

[40] M. Bourahla. Distributed CTL model checking. *IEE Proceedings - Software*, 152(6):297–308, 2005.

[41] L. Brim, I. Černá, and L. Hejtmánek. Parallel Algorithms for Detection of Negative Cycles. Technical Report FIMU-RS-2003-04, Faculty of Informatics, Masaryk University Brno, 2003.

[42] 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)*, volume 4144 of *LNCS*, pages 352–366. Springer, 2004.

[43] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.

[44] Lubos Brim and Jitka Žídková. Using Assumptions to Distribute Alternation Free [mu]-Calculus Model Checking. *ENTCS*, 89(1):17 – 32, 2003.

[45] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[46] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. de Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935 of *LNCS*, pages 181–200. Springer Verlag, 1995.

[47] 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.

[48] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 139–149. Society for Industrial and Applied Mathematics, 1995.

[49] Ming-Ying Chung and Gianfranco Ciardo. Saturation NOW. In *1st International Conference on Quantitative Evaluation of Systems (QEST 2004)*, pages 272–281. IEEE Computer Society, 2004.

[50] Ming-Ying Chung and Gianfranco Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE, 2006.

[51] Ming-Ying Chung and Gianfranco Ciardo. A Pattern Recognition Approach for Speculative Firing Prediction in Distributed Saturation State-Space Generation. *Electr. Notes Theor. Comput. Sci.*, 135(2):65–80, 2006.

[52] G. Ciardo, J. Gluckman, and D.M. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

[53] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 328–342. Springer, 2001.

[54] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-Based Symbolic Reachability Analysis Using Conjunctive and Disjunctive Partitioning. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 146–161. Springer, 2005.

[55] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? *CoRR*, abs/0912.2785, 2009.

[56] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.

[57] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[58] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.

[59] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix Sums and List Ranking. *Information and Computation*, 81(3):334–352, 1989.

[60] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science/Engineering/Math, July 2001.

[61] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[62] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 2.0,, 2009. `http://www.nvidia.com/object/cuda_develop.html`, June 2009.

[63] David L. Dill. The mur$\varphi$ verification system. In *Conference on Computer-Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 390–393. Springer-Verlag, July 1996.

[64] Matthew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Rahul Purandare. Parallel Randomized State-Space Search. In *International Conference on Software Engineering (ICSE 2007)*, pages 3–12. IEEE Computer Society, 2007.

[65] Stefan Edelkamp, Peter Sanders, and Pavel Šimeček. Semi-external LTL model checking. In *CAV '08: Proc. of the 20th international conference on Computer Aided Verification*, pages 530–542, Berlin, Heidelberg, 2008. Springer.

[66] Stefan Edelkamp and Damian Sulewski. Model Checking via Delayed Duplicate Detection on the GPU. Technical Report Technical Report 821, TU Dortmund, 2008. Presented on the 22nd Workshop on Planning, Scheduling, and Design PUK 2008.

[67] Stefan Edelkamp and Damian Sulewski. Parallel State Space Search on the GPU. In *International Symposium on Combinatorial Search (SoCS 2009)*, 2009.

[68] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9(1-2):105–131, 1996.

[69] J. Esparza and S. Schwoon. A note on on-the-fly verification algorithms. In *TACAS'05*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

[70] Sami Evangelista. Dynamic delayed duplicate detection for external memory model checking. In *SPIN '08: Proc. of the 15th international workshop on Model Checking Software*, pages 77–94, Berlin, Heidelberg, 2008. Springer.

[71] Jonathan Ezekiel, Gerald Luttgen, and Radu Siminiceanu. To Parallelize or to Optimize? *Advance access of Journal of Logic and Computation*, page exp006, 2009.

[72] L. K. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. In *Parallel and Distributed Processing*, volume 1800 of *LNCS*, pages 505–511. Springer, 2000.

[73] 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'2001)*, volume 2057 of *LNCS*, pages 216–234, Toronto, Canada, 2001. Springer-Verlag.

[74] H. Gazit and G. L. Miller. An Improved Parallel Algorithm That Computes the BFS Numbering of a Directed Graph. *Information Processing Letters*, 28(2):61–65, 1988.

[75] Jaco Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *SPIN*, pages 12–21, 1999.

[76] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *TACAS'04*, volume 2988 of *LNCS*, pages 205–219. Springer, 2004.

[77] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *International Conference on Management of Data (SIGMOD 06)*, pages 325–336. ACM, 2006.

[78] O. Grumberg, T. Heyman, and A. Schuster. Distributed Model Checking for $\mu$-calculus. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proc. 13th Conference on Computer-Aided Verification CAV01*, volume 2102 of *LNCS*, pages 350–362. Springer, 2001.

[79] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2005.

[80] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.

[81] Moritz Hammer and Michael Weber. "To Store or Not To Store" Reloaded: Reclaiming Memory on Demand. In Luboš Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2006.

[82] Hans Hansson and Bengt Jonsson. A Framework for Reasoning about Time and Reliability. In *IEEE Real-Time Systems Symposium*, pages 102–111, 1989.

[83] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.

[84] P. Harish, V. Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA, 2009.

[85] M. Harris. Optimizing Parallel Reduction in CUDA,. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`, March 2010.

[86] 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 *Proc. 8th Int. Workshop on Petri Net and Performance Models (PNPM'99), 8-10 October 1999, Zaragoza, Spain*, pages 12–21. IEEE Computer Society Press, 1999.

[87] Boudewijn R. Haverkort, Henrik Bohnenkamp, and Alexander Bell. Efficiency improvements in the evaluation of large stochastic petri nets. In Desel, J., Kemper, P., Kindler, E., and Oberweis, A., editors, *Forschungsbericht: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 55–61. Universität Dortmund,

Fachbereich Informatik, 1998. Published as Forschungsbericht: 5. Workshop Algorithmen und Werkzeuge für Petrinetze, number 694.

[88] Keijo Heljanko, Victor Khomenko, and Maciej Koutny. Parallelisation of the Petri Net Unfolding Algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2002.

[89] Tamir Heyman, Daniel Geist, Orna Grumberg, and Assaf Schuster. A Scalable Parallel Algorithm for Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, 21(3):317–338, 2002.

[90] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer Aided Verification, 12th International Conference*, volume 1855 of *LNCS*, pages 20–35. Springer, 2000.

[91] Fredrik Holmén, Martin Leucker, and Marcus Lindström. UppDMC – a distributed model checker for fragments of the $\mu$-calculus. In Lubos Brim and Martin Leucker, editors, *Proc. of the $3rd$ Workshop on Parallel and Distributed Methods for Verification*, volume 128/3 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2004.

[92] Viliam Holub and Petr Tůma. Streaming state space: A method of distributed model verification. In *1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 356–368. IEEE Computer Society, 2007.

[93] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[94] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[95] Gerard J. Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *ENTCS*, 198(1):3–16, 2008.

[96] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[97] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, page 6 pages. IEEE, 2008.

[98] Cornelia P. Inggs and Howard Barringer. CTL$^*$ model checking on a shared-memory architecture. *Electronic Notes in Computer Science*, 128(3):107–123, 2005.

[99] Shahid Jabbar and Stefan Edelkamp. I/O efficient directed model checking. In *Proc. of 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2005.

[100] Shahid Jabbar and Stefan Edelkamp. Parallel external directed model checking with linear I/O. In *Proc. of 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006) Charleston*, volume 3855 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2006.

[101] G. Jayachandran, V. Vishal, and V. S. Pande. Using massively parallel simulations and Markovian models to study protein folding: Examining the Villin head-piece. *Journal of Chemical Physics*, 124(6):903–914, 2006.

[102] Michael Jones and Eric Mercer. Explicit state model checking with Hopper. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 146–150. Springer, 2004.

[103] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In *Model Checking Software*, volume 3925 of *LNCS*, pages 126–145. Springer, 2006.

[104] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[105] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[106] W. Knottenbelt, P.G Harrison, M. Mestern, and P.S. Kritzinger. A Probabilistic Dynamic Technique for the Distributed Generation of Very Large State Spaces. *Performance Evaluation*, 35(1–4):127–148, Feb 2000.

[107] W. Knottenbelt, M. Mestern, P.G Harrison, , and P.S. Kritzinger. Probability, parallelism and the state space exploration problem. In R. Puigjaner, editor, *Tools'98*, volume 1469 of *LNCS*, pages 165–179. Springer Verlag, 1998.

[108] R. Korf. Best-First Frontier Search with Delayed Duplicate Detection. In *AAAI'04*, pages 650–657. AAAI Press / The MIT Press, 2004.

[109] R. Korf and P. Schultze. Large-Scale Parallel Breadth-First Search. In *AAAI'05*, pages 1380–1385. AAAI Press / The MIT Press, 2005.

[110] Lars Michael Kristensen and Thomas Mailund. Efficient path finding with the sweep-line method using external storage. In *ICFEM*, volume 2885 of *Lecture Notes in Computer Science*, pages 319–337. Springer, 2003.

[111] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. *Electronic Notes in Computer Science*, 128(3):19–34, 2005.

[112] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proc. TACAS'02*, 2002.

[113] Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *Formal Methods in Computer Aided Design (FMCAD 2010). To Appear*, 2010.

[114] Peter Lamborn and Eric A. Hansen. Layered duplicate detection in external-memory model checking. In *SPIN '08: Proc. of the 15th international workshop on Model Checking Software*, pages 160–175, Berlin, Heidelberg, 2008. Springer.

[115] Flavio Lerda and Riccardo Sisto. Distributed-memory Model Checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[116] Flavio Lerda and Willem Visser. Addressing Dynamic Issues of Program Model Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001)*, volume 2057 of *LNCS*, pages 80–102, Toronto, Canada, 2001. Springer-Verlag.

[117] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[118] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for Memory Hierarchies*. Springer, 2003.

[119] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A Parallel BDD Package. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 501–507. Springer, 1998.

[120] Kameshwar Munagala and Abhiram Ranade. I/O-complexity of graph algorithms. In *SODA '99: Proc. of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 687–694, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[121] D.M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-space Generation. *Journal of Parallel and Distributed Computing*, 47(2):122–131, 1997.

[122] D.M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-space Generation. Technical Report NASA/CR-2000-210082, NASA Langley Research Center, Hampton, USA, 2000.

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

[124] Doron Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.

[125] R. Pelánek. Fighting State Space Explosion: Review and Evaluation. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 37–52. Springer, 2009.

[126] Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in the disk based Mur$\varphi$ verifier. In *FMCAD*, pages 202–219, 2002.

[127] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[128] M. L. Puterman. *Markov Decision Processes-Discrete Stochastic Dynamic Programming*. John Wiley &Sons, New York, 1994.

[129] Rajeev K. Ranjan, Jagesh V. Sanghavi, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstation. In *International Conference on Computer Design (ICCD '96)*, pages 358–364. IEEE Computer Society, 1996.

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

[131] N. Rungta and E. G. Mercer. Generating Counter-Examples Through Randomized Guided Search. In *Model Checking Software (SPIN 2007)*, volume 4595 of *LNCS*, pages 39–57. Springer, 2007.

[132] On-the-fly, LTL model checking with SPIN.
URL: `http://spinroot.com/`.

[133] U. Stern and D. L. Dill. Parallelizing the mur$\varphi$ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.

[134] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the Mur$\varphi$ verifier. In *Computer Aided Verification. 10th International Conference*, pages 172–183, 1998.

[135] Tony Stornetta and Forrest Brewer. Implementation of an Efficient Parallel BDD Package. In *Proc. of Design Automation Conference (DAC'96)*, pages 641–644. ACM Press, 1996.

[136] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.

[137] M. Vardi. Automata-Theoretic Model Checking Revisited. In *VMCAI'07*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.

[138] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, Washington, DC, 1986.

[139] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 322–331. Computer Society Press, 1986.

[140] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

[141] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[142] S. Warren. Finding Strongly Connected Components in Parallel Using O(log2n) Reachability Queries. In *SPAA*, pages 146–151. ACM, 2008.

[143] Y. Zhang, D. Parker, and M. Kwiatkowska. Grid-enabled probabilistic model checking with PRISM. In *Proc. 4th All Hands Meeting Workshop (AHM'05)*, 2005.

[144] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689. AAAI Press / The MIT Press, 2004.

# Part II

# Collection of Articles

# Chapter 5

# Journal and Conference Papers

1. J. Barnat and P. Ročkai. Shared Hash Tables in Parallel Model Checking. *ENTCS*, 198(1):79–91, 2008.

2. J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):139–153, 2010.

3. K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

4. J. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

5. J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE Computer Society, 2009.

6. Jiří Barnat and Pavel Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

7. J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation Advance Access*, 2010.

8. J. Barnat, L. Brim, and P. Šimeček. I/O Efficient Accepting Cycle Detection. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.

9. J. Barnat, L. Brim, S. Edelkamp, D. Sulewski, and P. Šimeček. Can Flash Memory Help in Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 150–165. Springer-Verlag, 2008.

10. J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).*, volume 4963 of *LNCS*, pages 48–62. Springer, 2008.

11. J. Barnat, L. Brim, and P. Šimeček. Cluster-Based I/O Efficient LTL Model Checking. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 635–639. IEEE Computer Society, 2009.

12. J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Local Quantitative LTL Model Checking. In *Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *LNCS*, pages 53–68. Springer-Verlag, 2008.

13. J. Barnat, I. Černá, and J. Tůmová. Quantitative Model Checking of Systems with Degradation. In *Proceeding of the Sixth International Conference on Quantitative Evaluation of Systems (QEST 2009)*, pages 21–30. IEEE, 2009.

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 198 (2008) 79–91

**Electronic Notes in Theoretical Computer Science**

www.elsevier.com/locate/entcs

# Shared Hash Tables in Parallel Model Checking[1]

Jiří Barnat Petr Ročkai

*Faculty of Informatics, Masaryk University, Brno, Czech Republic*
barnat,xrockai@fi.muni.cz

**Abstract**

In light of recent shift towards shared-memory systems in parallel explicit model checking, we explore relative advantages and disadvantages of shared versus private hash tables. Since usage of shared state storage allows for techniques unavailable in distributed memory, these are evaluated, both theoretically and practically, in a prototype implementation. Experimental data is presented to assess practical utility of those techniques, compared to static partitioning of state space, more traditional in distributed memory algorithms.

*Keywords:* Hash tables, locking schemes, parallel

## 1 Introduction

Much of the extensive research on the parallelisation of model checking algorithms followed the distributed-memory programming model [5,4,12] and the algorithms were parallelised for networks of workstations, largely due to easy access to networks of workstations. Recent shift in architecture design toward multi-cores has intensified research pertaining to shared-memory paradigm as well.

A mostly straightforward transformation of distributed-memory algorithm into a shared-memory one, using several tailored techniques, is explored in [2]. In this paper, we intend to build on these results, further augmenting the selected distributed algorithms with extensions specific to shared-memory systems, especially those based on using a single shared storage for the explored graph.

For the experimental implementation, we have used DiVinE [3], specifically the multi-threaded, shared-memory version – as created for [2] – using the original DVE state-space generator. The code has been modified for the purposes of this paper.

[1] This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

doi:10.1016/j.entcs.2007.10.021

### *1.1 Shared-Memory Platform*

Since in the paper, we will work with several assumptions about the targeted hardware architecture, we will describe it briefly first.

We work with a model based on threads that share all memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Our working environment is POSIX, with its implementation of threads as lightweight processes. Switching contexts among different threads is cheaper than switching contexts among full-featured processes with separate address spaces, so using more threads than there are CPUs in the system incurs only a minor penalty.

**Critical Sections, Locking and Lock Contention.** In a shared-memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved through use of a "mutual exclusion device", so-called mutex. A thread wishing to enter a critical section has to lock the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behaviour. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

**Processor Cache: Locality and Coherence.** There are currently two main architectures in use for Level 2 cache. One is that each processing unit has its completely private Level 2 cache (for the Symmetric Multiprocessing case) or there is a shared Level 2 cache for a package of 2 cores (designs with a Level 2 cache shared among 4 cores are not commercially available as of this writing). In bigger shared-memory computer systems, it is usual to encounter split cache, since they often contain on the order of 8-64 cores attached to a single memory block. In recent hardware, the basic building units are dual-core CPUs with shared cache, but among the different units, the caches are still separate. This idiosyncrasy of these architectures has important effects on performance and these will be discussed later in more detail.

**Shared Memory Bus.** Since the memory in SMP machines is attached to a single shared memory bus, the RAM access from different processors needs to be serialized. This caps total memory throughput of the system and at some point, the available memory bandwidth becomes the bottleneck of computation. This is an important factor for memory-intensive workloads, to which model-checking definitely belongs.

### *1.2 Algorithms*

The algorithms used are not the main concern of this paper, but we nevertheless summarise OWCTY, as it was used in the implementation. Also, since we are using the algorithm in somewhat non-standard setting, we have slightly modified some of its non-vital aspects – more details on those modifications will be described in Section 3.3. Short description of the original algorithm follow.

The algorithm [8] is an extended enumerative version of the **One Way Catch Them Young Algorithm** [11]. The idea of the algorithm is to repeatedly remove vertices from the graph that cannot lie on an accepting cycle. The two removal rules are as follows. First, a vertex is removed from the graph if it has no successors in the graph (the vertex cannot lie on a cycle), second, a vertex is removed if it cannot reach an accepting vertex (a potential cycle the vertex lies on is non-accepting). The algorithm performs removal steps as far as there are vertices to be removed. In the end, either there are some vertices remaining in the graph meaning that the original graph contained an accepting cycle, or all vertices have been removed meaning that the original graph had no accepting cycles.

The time complexity of the algorithm is $\mathcal{O}(h \cdot m)$ where $h = h(G)$. Here the factor $m$ comes from the computation of elimination rules while the factor $h$ relates to the number of global iterations the removal rules must be applied. Also note, that an alternative algorithm is obtained if the rules are replaced with their backward search counterparts.

## 2 Hash Tables in Model Checking

One of the traditional approaches, when exploring the state-space of an implicitly specified model, is that the algorithm starts from the initial state and using a transition function, generates successors of every explored state. Visited states are stored in a hash-table, to facilitate quick insertion of newly visited states and quick lookup of states that already have been visited.

The usual approach in distributed algorithms is to partition the state space statically, using a partition function [7,9] (which is usually in turn based on a hash function over the state representation). This partition function unambiguously assigns each state to one of the computation nodes. Same approach can be leveraged in shared-memory computation, where each thread of control assumes ownership of a private hash table, and potentially also a private memory area for storing actual state representations.

The described configuration is often the only feasible option, when dealing with distributed memory system, since cross-node memory access has to be either manually simulated using message passing, or even if available, is prohibitively expensive.

However, the situation in shared-memory systems is somewhat different, since all processors (and therefore threads of control) share a single continuous block of local memory, with uniform accessibility from all the CPUs and/or cores. This gives us two new options, compared to situation in distributed environment, namely, if several hash tables are used, threads can look into tables they don't own, and second, probably more interesting option is to have a single shared hash table, used by all the threads.

### 2.1 Implementation

The threaded version of DiVinE implements an internal collision resolution hash table with quadratic probing. The table is dynamically-sized with exponential growth (i.e., the size of the table doubles every time more space is needed). Originally, the threshold triggering table growth has been set as half-full, which gives minimum overhead of 2 key-sized cells per valid item, where in our case the key is a single pointer. Growing the table starts with allocating a new, double-sized table, iterating over all entries in the old table and rehashing them into the new, bigger one. This is a linear-time operation, amortised over insertions into the table. However, this property may have more far-fetched consequences in a setting where the table is shared among multiple threads.

A somewhat different approach for triggering the growth of the table has been implemented as part of the work on shared hash tables. The conditions are now twofold, first is that table is 75% full, the second is that there have been too many collisions upon insert, where too many is defined as $32 + \mathrm{sqrt}(size)/16$. This parameter may be subject to further adujstment, although we haven't observed significant impact. This latter trigger produces more tighly packed tables, which may sometimes save time, especially since during the growth, all other processing is halted. Another possibility to reduce the number of grows is to increase the growth factor (this is an user-overridable setting and subject to empirical tuning).

### 2.2 Region Locking

There is a need for locking when multiple threads perform concurrent reads and updates of the table. Since the table is accessed very frequently, it is completely unfeasible to lock the whole table for each access, as this would lead to very high lock contention and, consequently, reduced performance. Therefore, a region locking scheme is devised, to only lock the region within which the update or lookup takes place. Special precautions are necessary for growing the table, since no updates at all are allowed during this window. The regions are fixed-size, so the number of regions grows linearly with the table size. There are two other options on how to organise locking, one being of fixed number of locks, which means the locking unit increases linearly with the hash table size, the second being a square-root based growth of both region size and number of locks.

Theoretical benefits of the first approach are that lock granularity and therefore contention should remain very low throughout program execution. Fixed number of locks makes competition for any given lock higher, although in theory, it should remain constant, as long as number of competing threads is constant. The square-root approach is a compromise between those two. All the methods are evaluated in the experimental section.

### 2.3 Lockless Shared Table

If implemented with no locking at all, an insertion may silently fail, i.e. it may be overwritten by a subsequent insert to a colliding position due to a race condition. However, this is not a fatal problem for reachability analysis, as observed in [17]. We have implemented a lockless hash table, but we have encountered severe scalability problems with large, statically sized tables (as opposed to dynamically growing tables). Since growing a lockless table is not implemented, this makes it hard to compare against the locking implementations, which can resize tables and therefore don't suffer from the large table problem. However, even lock-based tables, when statically sized, are highly detrimental to any scalability the system may be exhibiting. As of this writing, we haven't found the cause of the scalability issues with pre-sized tables, therefore more investigation is due.

## 3 State Space Partitioning

To distribute the workload of graph exploration (in case of safety checking) or cycle detection (in case of liveness checking), the state space is divided into parts, one for each of the worker threads (in the case of distributed computation, one for each cluster node).

### *3.1 Static Partitioning*

The original shared-memory implementation used a partitioning scheme coming directly from the distributed world. Each state is uniquely assigned to a thread, based solely on the state representation. This means, that every time a state is generated, it is assigned to the same thread. Consequently, each thread can maintain its private hash table, where it stores all states it owns. This has an important side-effect of the thread being able to operate on the table without resorting to locking or critical sections. Same goes for the auxiliary state data (like predecessor count in OWCTY elimination) – no locking is necessary.

Another benefit is highly efficient use of processor cache, by making the ratio of hash table size to processor cache size much more favourable, than in the case of shared hash table. This consequently reduces memory load and improves throughput.

### *3.2 Dynamic Partitioning*

The above static partitioning scheme suffers from high communication overhead, since as threads are added, number of cross-transitions (transitions that require inter-thread communication, because one of the states belongs to different thread than the other) grows rapidly.

A scheme using a different partitioning approach may be devised, when we are dealing with a single, shared table. Since the shared table allows any thread to lookup or update any state, it is no longer necessary to maintain the rule requiring each state to be unambiguously assigned to one of the threads. Instead, the thread that is examining a transition can decide on-the-fly whether to process it locally, or send it over to another CPU.

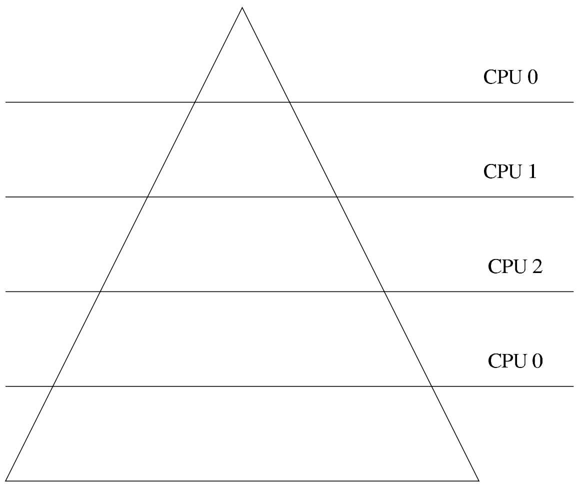


Fig. 1. Illustration of handoff partitioning scheme.

**Modular partitioning.** There are several possible approaches on how to partition the state space. A naïve implementation is to make every n-th transition a cross-transition, i.e. send it to a different thread. This makes for a great way to control the amount of cross transitions (and therefore explicit communication overhead). However, there are two problems with this approach. First, it reduces cache locality drastically, compared to that provided by static partitioning. In addition to losing the benefit of smaller hash tables (due to using a single big hash table), it also assigns states to threads virtually randomly, so it causes access to single state from different threads very often. This again reduces cache efficiency.

**Handoff partitioning.** This partitioning technique useful with DFS-based reachability analysis proposed in [13] is based on sending transitions to next thread when a certain "handoff depth" (depth of local DFS stack) is reached. This efficiently limits the amount of cross-transitions encountered, as they only appear every N levels of the pseudo-DFS tree, where N is the handoff depth or threshold. The actual threshold value is an option that needs to be empirically determined.

The technique has a much better state locality than the previous one, i.e. the chance that a given state is visited from a single thread several times is much higher. In Figure 1, a scheme of the resulting state distribution may be seen.

Another remarkable benefit of this scheme is the possibility to implement fairly efficient partial order reduction [15,16], as claimed in [13]. However, we have no such implementation and no comparison with other partial order reduction techniques, like [1,6].

**Shared queue.** Another possibility is to distribute states not using a partition function, but place them in a single shared BFS queue. This approach should achieve optimum load-balancing, although compromises may be necessary to strike a balance with locking overhead and contention.

### 3.3 Algorithm Impact

Through use of proper locking, all distributed algorithms can be used unmodified with shared hash table. However, the individual partitioning schemes place additional requirements on the algorithms, specifically on the visit order. The handoff technique requires a DFS stack and shared queue is specific to BFS.

Both the algorithms we have implemented are independent of order of visits, so can be run in both BFS and DFS order. These are reachability and OWCTY, although several other distributed algorithms share this property and could be therefore used in this setting. The parallel versions of Nested DFS [10] are not considered, since they do not use partitioning at all.

## 4 Experiments

Since there are no satisfactory profiling tools available for the kind of parallel workload we work with, we are mostly limited to measuring overall runtime of the algorithm implementations on various models using different parameters.

### 4.1 Methodology

The main testing machine we have used is a 16-way AMD Opteron 885 (8 CPU units with 2 cores each). All timed programs were compiled using gcc 4.1.2 20060525 (Red Hat 4.1.1-1) in 32-bit mode, using -O2. This limits addressable memory to 3GB, which was enough for our testing. The machine has 64GB of memory installed, meaning that none of the runs were affected by swapping.

For this paper, our main concern is speed and scalability, therefore we focus on these two parameters. Measurement was done using standard UNIX `time` command, which measures real and cpu times used by program. The real runtime is of particular interest, since this is the figure describing how long will the user wait for their results.

| Acronym | Description | Property (LTL formula) |
|---|---|---|
| *elevator* | Motivated by elevator promela model from distribution of SPIN. The cab controller chooses the next floor to be served as the next requested floor in the direction of the last cab movement. If there is no such floor then the controller consider the oposite direction. (3 floors) | If level 0 is requested, the cab passes the level without serving it at most once. $G(r0 \implies (\neg l_0 U(l_0 U (\neg l_0 U(l_0 U (l_0 \wedge open))))))$ |
| *leader* | Leader election algorithm based on filters. A filter is a piece of code that satisfy the two following conditions: a) if $m$ processes enter the filter, then at most $m/2$ processes exit; b) if some process enter the filter, then at least one of them exits. (5 processes) | Eventually a leader will be elected. $F(leader)$ |
| *rether* | Software-based, real-time Ethernet protocol whose purpose is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity Ethernet hardware. It is a contention-free token bus protocol for the datalink layer of the ISO protocol stack. (5 Nodes) | Infinitely many NRT actions of Node 0. $G(F(nact0))$ |
| *peterson* | Peterson's mutual exclusion protocol for N processes. (N=4) | Someone is in critical section infinitely many times. $G(F(SomeoneInCS))$ |
| *anderson* | Anderson's mutual exlusion protocol for N processes. (N=6) | $N/A$ |

Table 1
Models and verified properties.

All the models we have used are listed in Table 1 including the verified properties. The models come from the BEEM database [14] that contains the models in DiVinE-native modeling language.

## 4.2 Comparison of Partitioning Methods

First, we have measured reachability timings for the model peterson$_1$ using four approaches: BFS with static partitioning, BFS with modular partitioning, DFS with handoff partitioning and DFS with handoff partitioning and preallocated hash table (5 million cells, to accomodate the model easily). Also note that since the separate hash tables for BFS get smaller as the number of threads increases, the growth overhead drops slightly. We have also measured runtimes of OWCTY on the same model using analogical conditions. The results may be seen in Figure 2. From the figures, we see that for small number of cores, the dynamic partitioning schemes perform better, but are consistently "outscaled" by the statically partitioned BFS. Surprizingly, the modular partitioning scheme is not as far behind handoff as we have expected in some cases, although it still is the slowest and least scalable one.

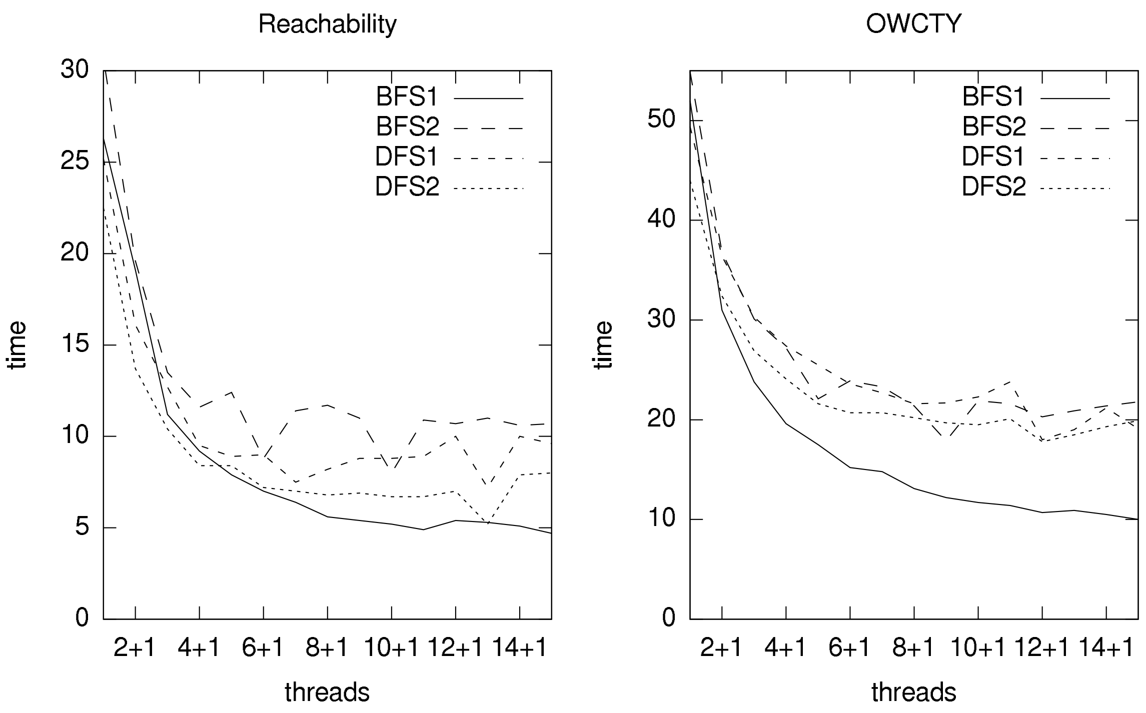


Fig. 2. Comparing scalability of reachability and OWCTY, using static and dynamic partitioning. BFS1 uses static partitioning, BFS2 uses modular partitioning, DFS1 uses handoff partitioning and finally DFS2 uses handoff with preallocation. Model used is $\text{peterson}_1$.

### *4.3 Effect of Handoff Threshold*

To determine the practical effect of handoff threshold on actual runtimes of the algorithms, we have measured runtimes of reachability and OWCTY with a matrix of parameter combinations using DFS and handoff partitioning. Figures 4 and 5 visualise data from the smaller model (peterson, on the order of 2 million states). We observe, that handoff does not affect the runtime significantly, unless set to very high – around 200, it starts to negatively affect scalability, being unable to provide sufficient load balancing.

We have also tried with a bigger model (anderson, on the order of 18 million states) using reachability. The results are available in Figure 6. Here, handoff depths up to 4096 seem to manage to spread the load evenly across threads, while at very low handoff (1-4), the number of cross-transitions slows the computation down significantly.

### *4.4 Effect of Locking Scheme*

In Figure 7, we present the behaviour of DFS reachability using various locking schemes, on top of a shared storage, using handoff partitioning (using default handoff depth of 50). All locking schemes were evaluated both using preallocated hash table and a growing hash table. From the picture, we see that the locking scheme basically does not affect runtime in any significant way.

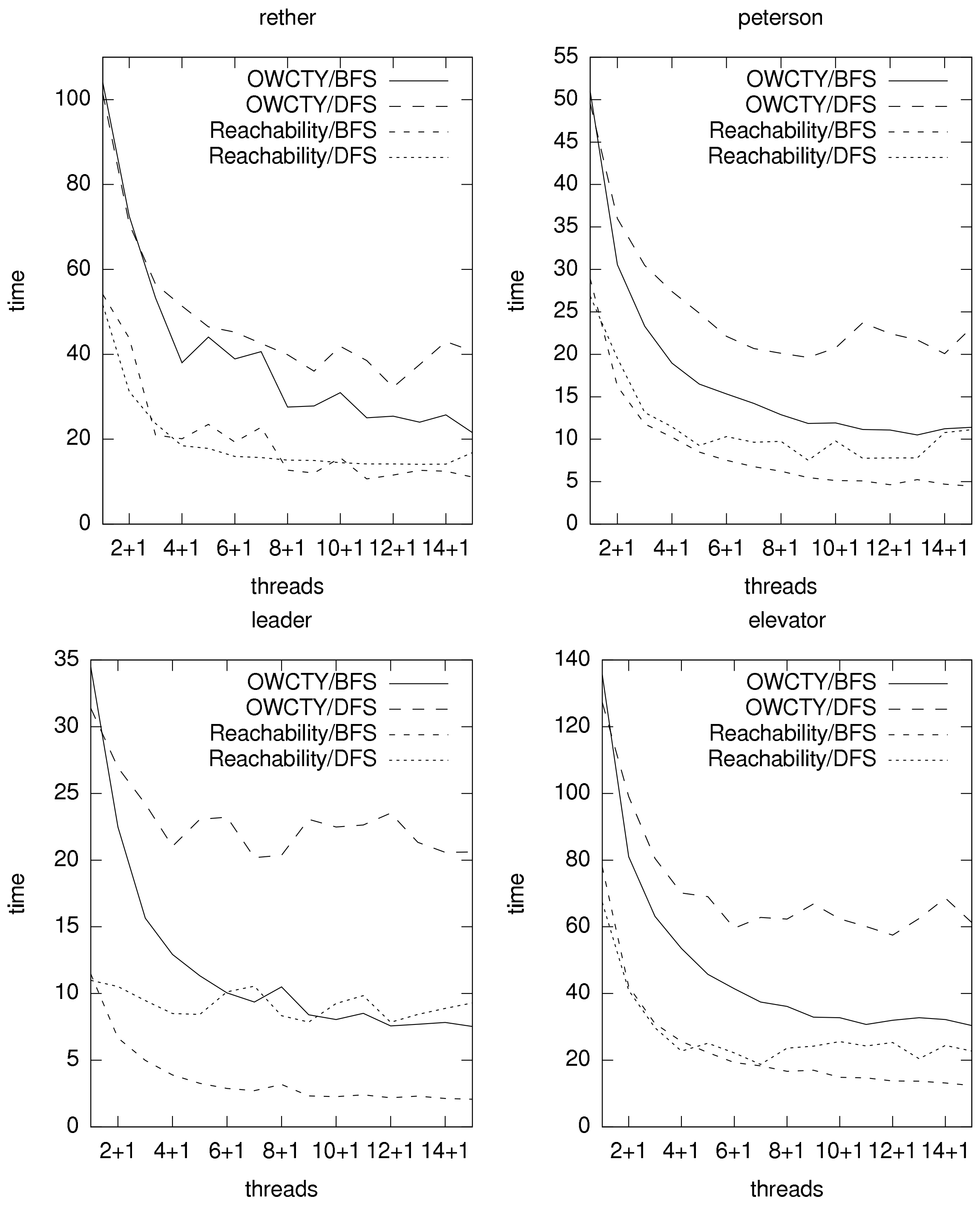


Fig. 3. Comparing scalability of reachability and OWCTY, using static and dynamic partitioning. BFS uses static partitioning, DFS1 uses handoff partitioning.

## 5 Conclusions

We have implemented several techniques dealing with use of shared hash tables in shared-memory parallel model checking. They have been compared, both theoretically and practically, to approaches known from distributed world.

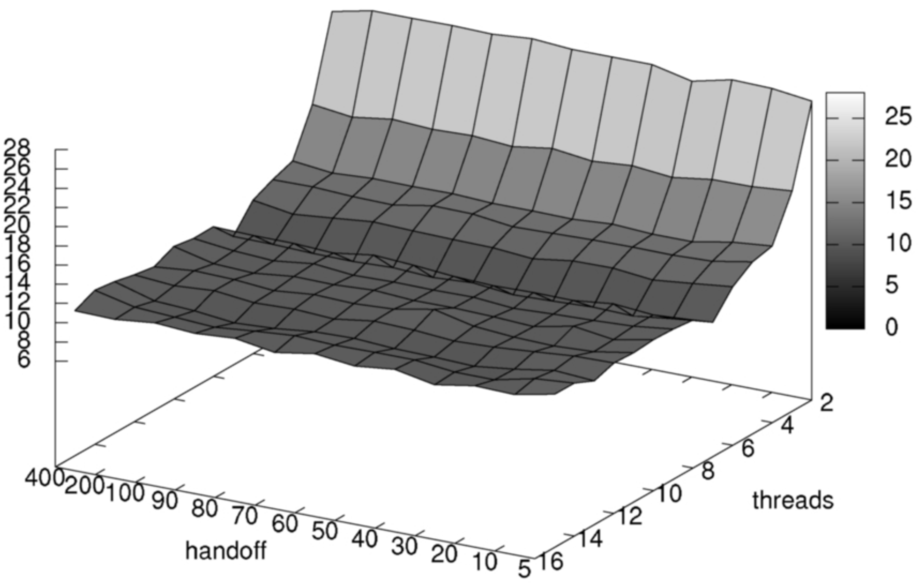


Fig. 4. Measuring effect of different handoff depths at runtimes and scalability of reachability on a small model ($\text{peterson}_1$). Note that the handoff axis is reversed!

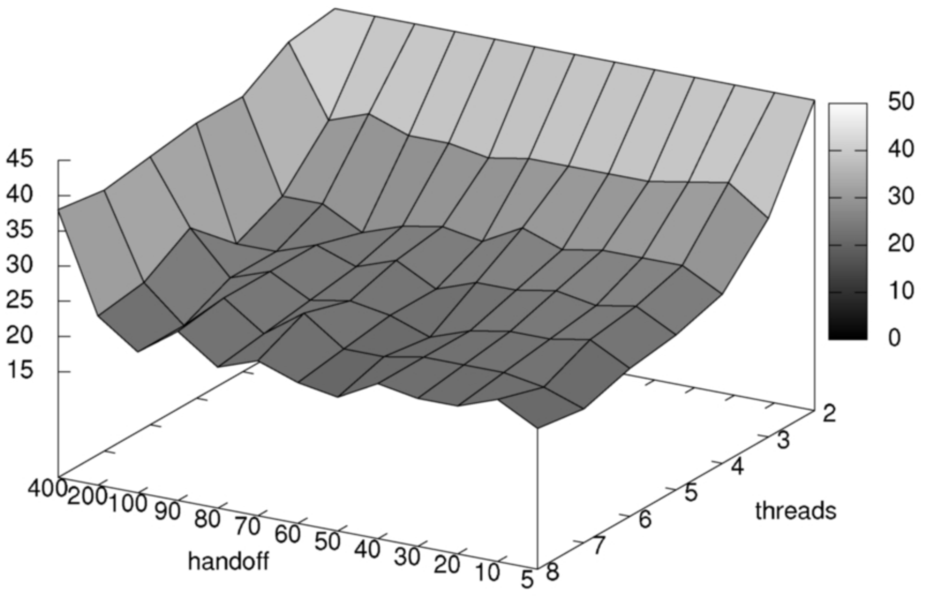


Fig. 5. Measuring effect of different handoff depths at runtimes and scalability of OWCTY on a small model ($\text{peterson}_1$). Note that the handoff axis is reversed!

In an environment with fairly low communication overhead, the different schemes did not vary as much as we have originally anticipated. The motivation behind the research was to improve performance and scalability of our parallel, shared-memory model checking platform based on DiVinE. However, the results have been less than convincing.

Although the schemes based on shared hash table, depth-first traversal and handoff partitioning have performed better on smaller number of threads (in the range of 1-8 threads), their utility in improving scalability over 8 cores is basically nonexistent. Breadth-first traversal with static partitioning, as used in distributed-

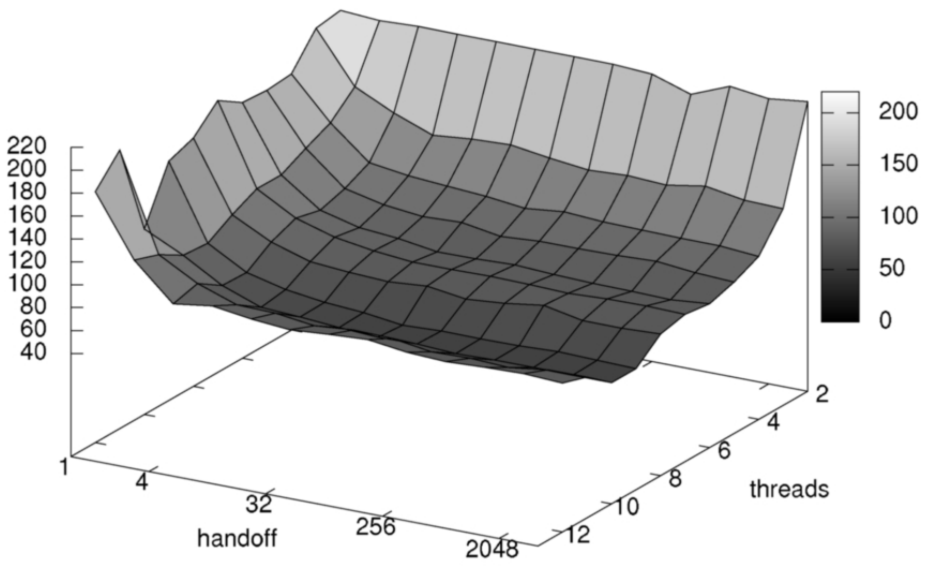
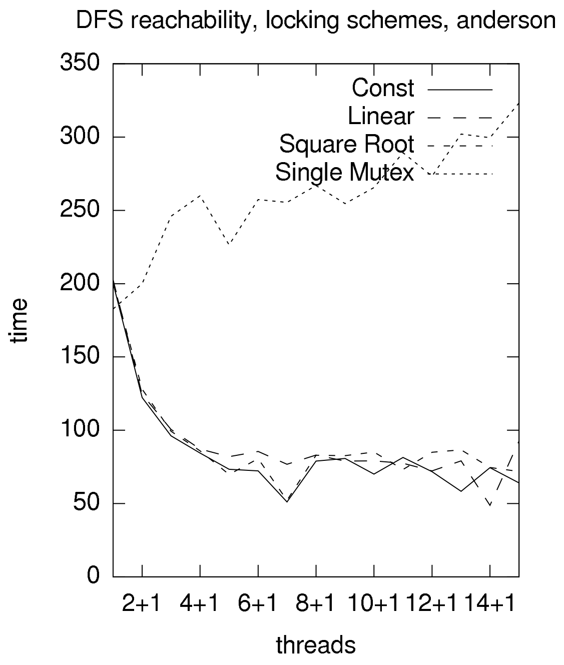


Fig. 6. Measuring effect of different handoff depths at runtimes and scalability of reachability. Big model (anderson). The handoff scale is logarithmic.



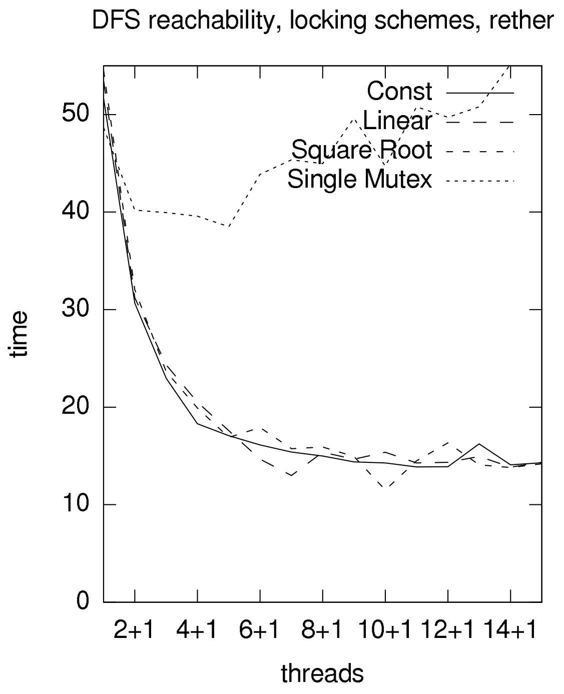


Fig. 7. Comparing locking methods on a reachability run over anderson and rether.

memory systems, out-scales them in these situations, by a not insignificant margin in some cases.

The main results therefore are, that communication overhead plays a role less important in scalability of shared-memory implementation, than previously believed. Second, that approaches known from distributed-memory architectures may be of practical utility to projects pursuing scalable shared-memory model-checking tool.

Since the results hint at a different source of limited scalability in shared memory systems than pure communication overhead, we will pursue further research on this problem. The candidates for investigation include suboptimal implementation (eg. false sharing or locking problems) and hardware architecture limitations.

We have already identified and mitigated several problems impeding scalability in various scenarios, including false sharing and excessive thread migration among available cores caused by kernel scheduler. The general pattern we have observed is, that improvements in scalability are gradual and that there is no proverbial silver bullet, that would solve all the scalability issues at once.

## References

[1] Barnat, J., L. Brim and J. Chaloupka, *From Distributed Memory Cycle Detection to Parallel LTL Model Checking*, Electronic Notes in Theoretical Computer Science **133** (2005), pp. 21–39.

[2] Barnat, J., L. Brim and P. Ročkai, *Scalable Multi-Core LTL Model-Checking*, in: *Proc. of SPIN 2007, to appear*, LNCS **4595** (2007), pp. 197–203.

[3] Barnat, J., L. Brim, I. Černá, P. Moravec, P. Ročkai and P. Šimeček, *DiVinE – A Tool for Distributed Verification (Tool Paper)*, in: *Computer Aided Verification*, LNCS **4144/2006** (2006), pp. 278–281.

[4] Behrmann, G., T. S. Hune and F. W. Vaandrager, *Distributed timed model checking — how the search order matters*, in: *Proc. 12th Conference on Computer-Aided Verification CAV00*, LNCS **1855** (2000), pp. 216–231.

[5] Brim, L. and J. Barnat, *Distribution of explicit-state ltl model-checking*, in: T. Arts and W. Fokkink, editors, *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, Electronic Notes in Theoretical Computer Science **80** (2003).

[6] Brim, L., I. Černá, P. Moravec and J. Šimša, *Distributed Partial Order Reduction of State Spaces*, in: *3rd International Workshop on Parallel and Distributed Methods in verifiCation*, 2004.

[7] Caselli, S., G. Conte and P. Marenzoni, *Parallel state space exploration for GSPN models*, in: G. de Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, LNCS **935** (1995), pp. 181–200.

[8] Černá, I. and R. Pelánek, *Distributed explicit fair cycle detection (set based approach)*, in: T. Ball and S. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop*, Lecture Notes in Computer Science **2648** (2003), pp. 49 – 73.

[9] Ciardo, G., J. Gluckman and D. Nicol, *Distributed State Space Generation of Discrete-State +Stochastic Models*, INFORMS Journal on Computing **10** (1998), pp. 82–93.

[10] Courcoubetis, C., M. Vardi, P. Wolper and M. Yannakakis, *Memory-Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design **1** (1992), pp. 275–288.

[11] Fisler, K., R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang, *Is there a best symbolic cycle-detection algorithm?*, in: *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2031** (2001), pp. 420–434.

[12] Garavel, H., R. Mateescu and I. Smarandache, *Parallel State Space Construction for Model-Checking*, in: M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model +Checking of Software (SPIN'2001)*, LNCS **2057** (2001), pp. 216–234.
URL citeseer.nj.nec.com/474094.html

[13] Holzmann, G., *The Design of a Distributed Model Checking Algorithm for SPIN*, in: *FMCAD, Invited Talk*, 2006.

[14] Pelánek,
R., *BEEM: BEnchmarks for Explicit Model checkers*, http://anna.fi.muni.cz/models/index.html (2007).

[15] Peled, D., *Ten years of partial order reduction*, in: *Proceedings of the 10th International Conference on Computer Aided +Verification* (1998), pp. 17–28.

[16] Valmari, A., *Stubborn set methods for process algebras*, in: *Proceedings of the DIMACS workshop on Partial order methods in +verification* (1997), pp. 213–231.

[17] Weber, M., private communication (2007).

Int J Softw Tools Technol Transfer (2010) 12:139–153
DOI 10.1007/s10009-010-0136-z

SPIN 07

# Scalable shared memory LTL model checking

**J. Barnat · L. Brim · P. Ročkai**

Published online: 3 February 2010

**Abstract** Recent development in computer hardware has brought more widespread emergence of shared memory, multi-core systems. These architectures offer opportunities to speed up various tasks—model checking and reachability analysis among others. In this paper, we present a design for a parallel shared memory LTL model checker that is based on a distributed memory algorithm. To improve the scalability of our tool, we have devised a number of implementation techniques which we present in this paper. We also report on a number of experiments we conducted to analyse the behaviour of our tool under different conditions using various models. We demonstrate that our tool exhibits significant speedup in comparison with sequential tools, which improves the workflow of verification in general.

**Keywords** Parallel · Shared memory · Formal verification · LTL model checking

## 1 Introduction

With the arrival of 64-bit technology, the traditional space limitations in formal verification are diminishing in importance. The time required for a verification run is becoming an important bottleneck instead. This naturally raises interest in using parallelism to improve the performance of many formal verification tools.

Much of the extensive research on the parallelisation of model checking algorithms followed the *distributed memory* programming model which stemmed from the necessity to fight the memory constraints of a single computer system. Networks of workstations are easily accessible and they provide the desired computational power, aggregated memory in particular. Parallel, distributed memory techniques have been successfully applied to explicit-state (or enumerative) model checking [2,4,41], symbolic model checking [25,26], analysis of stochastic [27] and timed [9] systems, equivalence checking [12] and other related problems [10,13,22]. For a survey on parallel LTL model checking algorithms we refer to [6].

A recent shift in architecture design toward multi-cores with large amounts of local RAM has intensified research pertaining to *shared memory* paradigm as well. In [29] Holzmann and Bosnacki proposed an extension of the SPIN model checker for multicore machines. They suggested two different parallel algorithms for verification of safety and liveness properties. While the algorithm for checking safety properties scales well to N-core systems, the algorithm for liveness checking, which is based on SPIN's original nested depth-first search (DFS) algorithm, has scalability limited to dual-core systems. It is still an open problem to do scalable verification of general liveness properties on N-cores with time complexity linear in the size of the product automaton.

A different approach to shared memory model checking is presented in [31], based on CTL* translation to Hesitant Alternating Automata. The proposed algorithm uses a so-called non-emptiness game for deciding validity of the original formula and is, therefore, largely unrelated to the algorithms based on fair cycle detection.

This work was partially supported by the Grant Agency of Czech Republic grants No. 201/09/1389 and No. 201/09/P497 and the Academy of Sciences grant No. 1ET408050503, and P. Ročkai partially by Red Hat, Inc.

J. Barnat (✉) · L. Brim · P. Ročkai
Faculty of Informatics, Masaryk University,
Brno, Czech Republic
e-mail: barnat@fi.muni.cz

L. Brim
e-mail: brim@fi.muni.cz

P. Ročkai
e-mail: xrockai@fi.muni.cz

In this paper, we propose a design for a parallel shared memory model checking tool that is based on known distributed memory algorithms. For the prototype implementation, we considered the algorithm by Černá and Pelánek [18]. This algorithm is linear for properties expressible as weak Büchi automata, which comprise the majority of LTL properties encountered in practice. Although the worst-case complexity is quadratic, the algorithm exhibits very good performance with real-life verification problems. To achieve good scalability, we have devised several implementation techniques, as presented in this paper, and applied them to the algorithm. We expect that application of the proposed implementation approaches to other distributed memory algorithms for LTL model checking may bring about similar improvements in scalability on multi-core systems.

We have published a tool based on the presented results, under the name DiVinE Multi- Core. Its full source code is available from [3], together with the instructions on compilation and usage. The tool is meant to be used on shared memory multi-processor and multi-core computers, and it is capable of performing reachability analysis and OWCTY-based LTL model checking both on N-core systems.

In Sect. 2 we summarise the existing parallel algorithms for LTL model checking (*accepting cycle detection*). In Sect. 3, we give an overview of implementation techniques that were applied to multi-core implementations of the selected algorithms. In Sect. 4, we give a broad selection of experimental data on scalability and performance. A comparison to the most recent multi-core-capable version (5.1.4) of SPIN is given as well. Moreover, the effect of several optimisations on both performance and scalability is measured.

## 2 Parallel LTL model-checking algorithms

An efficient parallel solution to many problems often requires approaches radically different from those used to solve the same problems sequentially. Classical examples are list rankings, connected components, and depth-first search in planar graphs. In the area of LTL model checking, the best-known enumerative *sequential* algorithms based on fair cycle detection are the *Nested DFS* algorithm [20,30] (implemented, e.g., in the model checker SPIN [28]) and *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into strongly connected components (SCCs) [39]. These optimal sequential algorithms differ in their space requirements, length of the counter-example produced, and other aspects. For a recent survey we refer to [40]. The main idea of the Nested DFS algorithm is to use two interleaved searches to detect reachable accepting cycles. The first search discovers accepting states while the second one, the nested one, checks for self-reachability. Several modifications of the algorithm have been suggested to remedy some of its disadvantages. References [23,24] have proposed modifications of Tarjans algorithm, whose common feature is that they recognise an accepting cycle as soon as all transitions on the cycle are explored. For a survey and detailed comparison we refer to [37].

However, both types of algorithms rely on inherently sequential depth-first search (DFS) postorder. Unfortunately, it is not known how the DFS postorder can be computed efficiently in parallel. Therefore, it is difficult to adapt known DFS-based LTL model checking algorithms to parallel architectures. This is also the reason why the parallel algorithm of SPIN is limited to dual-core platforms. In particular, the algorithm performs two nested depth-first searches while the outer search must preserve the DFS postorder on backtracked states. As a result, the outer search cannot be executed in parallel and the algorithm cannot efficiently use more than two cores.

Consequently, different techniques and algorithms are needed for a parallel verification. Unlike the LTL model checking, the reachability analysis is a verification problem for which an efficient parallel solution is available. The reason is that the exploration of the state space is independent of the search order. In the following, we sketch four parallel algorithms for enumerative LTL model checking that are, more or less, based on performing multiple parallel reachability procedures to detect a reachable accepting cycle. The reader is kindly asked to consult the original sources for the details.

MAP. The main idea of the *Maximal Accepting Predecessor Algorithm* [14,16] is based on the fact that every accepting vertex lying on an accepting cycle is its own predecessor. An algorithm that is directly derived from the idea, would require expensive computation as well as space to store all proper accepting predecessors of all (accepting) vertices. To overcome this obstacle, the MAP algorithm stores only a single representative of all proper accepting predecessor for every vertex. The representative is chosen as the *maximal accepting predecessor* accordingly to a presupposed linear ordering $\prec$ of vertices (given, for example, by their memory representation). Clearly, if an accepting vertex is its own maximal accepting predecessor, it lies on an accepting cycle. Unfortunately, it can happen that all the maximal accepting predecessor lie outside accepting cycles. In that case, the algorithm removes all accepting vertices that are maximal accepting predecessors of some vertex, and recomputes the maximal accepting predecessors. This is repeated until an accepting cycle is found, or there are no more accepting vertices in the graph.

The time complexity of the algorithm is $\mathcal{O}(a^2 \cdot m)$, where $a$ is the number of accepting vertices and $m$ is the number of edges. One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertices used by the algorithm. Computing the optimal ordering is, however, difficult to parallelise; hence, heuristics for computing a suitable vertex ordering are used. □

OWCTY. The next algorithm [18] is an extended enumerative version of the *One Way Catch Them Young Algorithm* [21]. The idea of the algorithm is to repeatedly remove vertices from the graph that cannot lie on an accepting cycle. The two removal rules are as follows: First, a vertex is removed from the graph if it has no successors in the graph (the vertex cannot lie on a cycle), and second, a vertex is removed if it cannot reach an accepting vertex (a potential cycle the vertex lies on is non-accepting). The algorithm performs removal steps as far as there are vertices to be removed. In the end, either there are some vertices remaining in the graph meaning that the original graph contained an accepting cycle, or all vertices have been removed meaning that the original graph has no accepting cycles.

The time complexity of the algorithm is $\mathcal{O}(h \cdot m)$, where $h$ is the height of the SCC quotient graph. Here, the factor $m$ comes from the computation of elimination rules while the factor $h$ relates to the number of global iterations required for application of the removal rules. Also note that an alternative algorithm is obtained if the rules are replaced with their backward search counterparts.

OWCTY is the algorithm we have chosen as a primary one for the tool, in part due to its favourable time complexity and also thanks to its performance and scaling behaviour observed in practice. This choice does not preclude the use of the presented techniques with any other distributed memory algorithm, although we do not expect their performance to be an improvement over OWCTY. □

NEGC. The idea behind the *Negative Cycle Algorithm* [15] is a transformation of the LTL model checking problem to the problem of negative cycle detection. Every edge of the graph outgoing from a non-accepting vertex is labelled with 0 while every edge outgoing from an accepting vertex is labeled with $-1$. Clearly, the graph contains a negative cycle if and only if it has an accepting cycle.

The algorithm exploits the *walk to root* strategy to detect the presence of a negative cycle. The strategy involves construction of the so-called *parent graph* that keeps the shortest path to the initial vertex for every vertex of the graph. The parent graph is repeatedly checked for the existence of the path. If the shortest path does not exist for a given vertex, then the vertex is part of a negative (and therefore accepting) cycle. The worst-case time complexity of the algorithm is $\mathcal{O}(n \cdot m)$, where $n$ is the number of vertices and $m$ is the number of edges. □

BLEDGE. An edge $(u, v)$ is called a *back-level edge* if it does not increase the distance of the target vertex $v$ form the initial vertex of the graph. The key observation connecting the cycle detection problem with the back-level edge concept, as used in the *Back-Level Edges Algorithm* [1], is that every cycle contains at least one back-level edge. Back-level edges are therefore used as triggers to start a procedure that checks whether the edge belongs to an accepting cycle. However, this is too expensive to be done completely for every back-level edge. Therefore, several improvements and heuristics are suggested and integrated within the algorithm to decrease the number of tested edges and speed up the cycle test.

The BFS procedure which detects back-level edges runs in time $\mathcal{O}(m + n)$. In the worst case, each back-level edge may trigger a search for accepting cycle, which requires linear time $\mathcal{O}(m + n)$ as well. Since there is at most $m$ back-level edges, the overall time complexity of the algorithm is $\mathcal{O}(m.(m + n))$. □

All the algorithms allow for a scalable parallel implementation based on partitioning the graph (its vertices) into disjoint parts. Suitable partitioning is an important factor in overall efficiency of the parallelisation.

One particular technique that is specific to automata-based LTL model checking is *cycle locality preserving* problem decomposition [5,32]. The graph (product automaton) originates from a synchronous product of the property and system automata. Hence, vertices of product automaton graph are ordered pairs. An interesting observation is that every cycle in a product automaton graph arises from cycles in system and property automaton graphs. Let $A$, $B$ be Büchi automata and $A \otimes B$ their synchronous product. If $\mathcal{C}$ is a SCC in the automaton graph of $A \otimes B$, then the $A$-projection of $\mathcal{C}$ and the $B$-projection of $C$ are (not necessarily maximal) SCCs in the automaton graphs of $A$ and $B$, respectively.

As the property automaton is derived from the LTL formula to be verified, it is typically quite small and can be pre-analysed. In particular, it is possible to identify all SCCs of the property automaton graph. A partition function may then be devised that respects SCCs of the property automaton and therefore preserves cycle locality. The partitioning strategy is to assign all vertices that project to the same SCC of the property automaton graph to the same subproblem. Since no cycle is split among different subproblems, it is possible to employ a localised Nested DFS algorithm to perform local accepting cycle detection simultaneously.

Moreover, further interesting information can be drawn from the property automaton graph decomposition. Maximal SCCs can be classified into three categories:

**Type F:** (*Fully Accepting*) Any cycle within the component contains at least one accepting vertex. (There is no non-accepting cycle within the component.)

**Type P:** (*Partially Accepting*) There is at least one accepting cycle and one non-accepting cycle within the component.

**Type N:** (*Non-Accepting*) There is no accepting cycle within the component.

Realising that a vertex of a product automaton graph is accepting only if the corresponding vertex in the property automaton graph is accepting, it is possible to characterise the types of the SCCs in the product automaton, based on the types of the corresponding components in the property automaton. This classification of components into three distinct types, $N$, $F$, and $P$, can be used to gain additional improvements that may be incorporated into the algorithms given above.

Specifically, the OWCTY algorithm needs to use this additional information for it to be linear for all weak graphs. However, for practically encountered property automata, this is not strictly necessary, as the algorithm exhibits linear behaviour even without this modification.

## 3 Implementation techniques

It is a well-known fact that a distributed memory, parallel algorithm can be straightforwardly transformed into a shared memory one. However, there are several inefficiencies involved in this direct translation. The shared memory architecture has several traits which may offer advantages in real-world performance of such implementations. In this section, we present our approaches to the challenges of shared memory architecture and its specific characteristics. We will briefly describe the techniques introduced in [2], concerning communication, memory allocation, and termination detection, and we will show their application to the OWCTY algorithm described in Sect. 2. In addition to this, we introduce some of our latest results regarding implementation. First of all though, let us describe the target platform in more detail.

### 3.1 Shared-memory platform

Since we will work with several assumptions about the targeted hardware architecture in the paper, we will briefly describe the platform first.

Our working environment is POSIX threads. Therefore, we work with a model based on threads that share all the memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Contrast this with SPIN 5.1 that employs processes and inter-process communication channels to handle parallel computation.

*Critical Sections, Locking and Lock Contention.* In a shared memory setting, an access to a memory place that is used by multiple threads has to be controlled; otherwise, a race condition may occur. This is generally achieved by using a “mutual exclusion device”, the so-called mutex. A thread wishing to access the memory place has to lock the associated mutex to guarantee the exclusiveness. The access is then performed in the so-called critical section. Locking procedure may block the calling thread if the mutex is locked by some other thread. Such a situation is called a resource or lock contention. It occurs whenever two or more threads happen to need to access the same critical section (and therefore lock the same mutex) at the same time. If the critical sections are long or they are entered very often, the contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

*Processor Cache: Locality and Coherence.* There are currently two main architectures in use for Level 2 cache. One is that each processing unit has its completely private Level 2 cache (for the Symmetric Multiprocessing case), or there is a shared Level 2 cache for a package of 2 cores. In bigger shared memory computer systems, it is usual to encounter split cache, since they often contain on the order of 8–64 cores attached to a single memory block. In recent hardware, the basic building units are dual-core CPUs with shared cache, but among the different units, the caches are still separate.

Due to coherence requirements, a read of data from thread B subsequent to a write of the date from thread A may (and usually does) incur a significant penalty. Since the cache often works with smallest units of 128 or more bytes long (the so-ca cache lines), various pieces of data may share a single cache line, if they are adjacent in memory. If different threads access some data on a single cache line very often the performance of the computation may suffer dramatically. Note that this is also the case if the data accessed by two threads are disjoint, but close enough to occupy the same cache line. Such a situation is spoken about as of false sharing. Much more detailed study of these phenomenon may be found in, e.g. [38].

### 3.2 Implementing algorithms in shared memory

Whenever an algorithm is about to be implemented in a shared memory setting, all the technical details laid out in previous paragraphs must be taken into account. Recall that our goal is to adapt distributed memory algorithms to shared memory environment and to achieve a scalable implementation. The scalability is inversely proportional to communication overhead and its growth with increasing number of threads. Therefore, the techniques we designed were aimed to reduce communication overhead by exploiting traits of shared memory systems that are not available in distributed memory environment. However, keeping in mind the possibility to scale beyond shared memory systems, we tried to keep the implementation in a shape that would make a combined tool to work efficiently on clusters of multi-CPU machines achievable.

When we venture into a strictly shared memory implementation, one may pose a question whether a different approach of using a standard serial algorithm modified to allow parallelisation at lower levels of abstraction would give a scalable, efficient program for multi-CPU and/or multi-core systems. Our efforts at extracting such a micro-parallelism in our codebase have been largely fruitless, due to high synchronisation cost relative to the amount of work we were able to perform in parallel. An example of such micro-parallel approach would be to implement a parallel successor generator, where there is certain independence of the sub-tasks involved, but these sub-tasks are very small and on current hardware, it is faster to perform them in sequence than in parallel.

In the following sections, we explore the possibilities to build on existing distributed memory approaches, in the vein of statically partitioned graphs, reducing the overhead using idioms only possible due to locality of memory.

## 3.3 Communication

Generally, in a distributed computation, all communication is accomplished by passing messages—e.g. using a library like MPI for cluster message passing. However, in communication-intensive programs, or those sensitive to communication delay, using general-purpose message passing may be fairly inefficient.

In shared memory, most of the communication overhead can be eliminated by using more appropriate communication primitives, like high-performance, contention- and lock- free FIFOs (First In, First Out queues). We have adopted a variant of the two-lock algorithm—a decent compromise between performance on the one hand and simplicity and portability on the other—presented in [35]. Our modifications involve improved cache-efficiency and only use a single write-lock, instead of a pair of locks, one for reading and one for writing, since there is ever only one thread reading, while there may be several trying to write.

Representation and pseudo-code for enqueue and dequeue algorithms are found in Figs. 1, 2 and 3, respectively. The correctness, linearisability (atomicity) and liveness proofs as given in [35] are straightforwardly adapted to our implementation and thus left out.

Originally, every thread involved in the computation owned a single instance of the FIFO and all messages for this thread are pushed onto this single queue (there comes the need for the write lock).

**Fig. 1** FIFO representation

```
type FIFO of T:
    type Node:
        $buffer$: array of T
        $next$: pointer to Node
        $read$, $write$: integer
    $nodeSize$: integer (size of $buffer$)
    $head$, $tail$: pointer to Node
    $writeLock$: mutex
```

```
Require: $f$ is a FIFO of T instance, $x$ of type T is an element to enqueue
Ensure: $f$ contains $x$ as its last element
  lock( $f.writeLock$ )
  if $f.tail.write = f.nodeSize$ then
    $t \leftarrow$ newly allocated Node, all fields 0
  else
    $t \leftarrow f.tail$
  $t.buffer[t.write] \leftarrow x$
  $t.write \leftarrow t.write + 1$
  if $f.tail \neq t$ then
    $f.tail.next = t$
    $f.tail = t$
  unlock( $f.writeLock$ )
```

**Fig. 2** FIFO enqueue

```
Require: $f$ is a non-empty FIFO instance
Ensure: front element of $f$ is dequeued and then returned
  if $f.head.read = f.nodeSize$ then
    $f.head \leftarrow f.head.next$
  $f.head.read \leftarrow f.head.read + 1$
  return $f.head.buffer[f.head.read - 1]$
```

**Fig. 3** FIFO dequeue

However, in communication-intensive workloads (like our parallel model checking algorithms), the write lock has been observed to be a point of contention, creating a bottleneck even when only 4 CPU cores were involved.

Although there is also a completely lock-free design described in [35], we have opted for a matrix-style communication primitive, with a private FIFO for each pair of communicating threads. This is partially motivated by the use of atomic compare and swap instructions in the lock-free queue design, which are relatively expensive compared with regular memory access. Moreover, even when lock contention is removed, the lower level contention for a single memory location on the CPU level is likely to be a reason of concern. Moreover, for our use-case, all of the mentioned issues can be addressed by using a larger number of queues without incurring any significant penalties.

Alternatives to our implementation, which may be more appropriate in different settings, include a ring-buffer FIFO implementation (if there is a bound on the amount of in-flight data known beforehand, the ring-buffer implementation may be more efficient) and possibly an algorithm based on swapping incoming and outgoing queues (which could be easily implemented as a pointer swap). The latter gives results comparable with the described FIFO method, although the code and locking behaviour is much more complex and error-prone, which made us opt for the simpler FIFO implementation.

## 3.4 Memory allocation

In a distributed computation, every process has simply its own memory which it fully manages. In a shared memory, however, we prefer to manage the memory as a single shared area, since an equal partitioning of available memory and separate management may fall short of efficient resource usage. However, this poses some challenges, especially in allocation-intensive environment like ours.

*Efficient allocation and deallocation routines.* Since the workload we are facing facilitates large amounts of fixed-size allocations and deallocations, it appears natural, to implement tailored allocation and deallocation routines.

A very simple O (1) memory pool has been devised, optimised for many allocations of a limited set of sizes. Of course, from time to time it needs to obtain memory from the system and this operation is not constant-time; however, it is at most linear in the block size and therefore amortises over the individual allocations as well (given a fixed allocation size).

Each thread has its own private pool and therefore the implementation is lock-free. This is possible since most operations are thread-local: the remaining case of cross-thread deallocation is discussed in the following.

*Concurrent allocation and deallocation.* First, a naïve approach of protecting the allocation routines with a simple mutual exclusion is highly prone to resource contention. Fortunately, modern general-purpose allocator implementations refrain from this idea and have a generally non-contending behaviour on allocation. However, releasing memory back for reuse is more complex to achieve without introducing contention, in a setting where it is often the case that thread other than the one allocating the chunk needs to release it.

There are known general-purpose solutions to this problem, e.g. [34]; however, they are currently not in widespread use in general-purpose allocators. Therefore, when relying on the system's allocator, we have to refrain from the aforementioned pattern of releasing memory from different than allocating thread, in order to avoid contention and the accompanying slowdown.

The message-passing implementation we employ is pointer-based; in other words, the message sent is only a pointer and the payload (actual interesting message content) is allocated on the shared heap, and it may be either reused or released by the receiving thread. Observe, however, that releasing the associated memory in the receiving thread will introduce the situation which we are trying to avoid.

Therefore, the allocation routines handle these cases differently. Instead of manipulating the memory pool of a different thread, the memory (allocated in a different thread) is appended to the freelist of the current thread. Thanks to the workload distribution scheme employed, this approach is very much feasible and does not introduce significant memory overhead. Moreover, it is correct, since the memory handed over to the new thread is never again examined by the original thread. We have dubbed the technique "memory stealing", since the releasing thread "steals" the memory from its previous owner.

*General purpose memory allocation.* Since apart from the state allocation and deallocation, there are several important memory-intensive routines (one example being the FIFOs, another is the model parser and interpreter) in the model checker, which do not exhibit the behaviour described above, we also need a high-performance, general-purpose memory allocator. Moreover, the pool allocator described needs to obtain memory blocks somewhere as well.

We have opted for Emery Berger's excellent HOARD multi-threaded memory allocator [11]. Apart from having very good performance and scalability properties, HOARD strives to avoid heap layouts leading to false sharing, further improving performance.

## 3.5 Termination detection

Since our algorithms rely on work distribution among several largely independent threads, we need an algorithm for shared memory termination detection. Similar to the distributed memory setting, our algorithm should introduce minimal overhead and avoid as much serialisation as possible.

One possible solution is presented in [33]. The solution avoids locking at all; however, it requires that the system provides an enqueue-with-wakeup primitive. We decided not to follow this lock-free approach as we were able to achieve quite satisfactory solution much easily employing primitives available in POSIX Thread API. In particular, the API offers a mutex implementation that allows threads to use the mutex in a lock-or-fail manner, as opposed to the standard lock-or-wait, which is usually employed for protecting critical sections. We can leverage this mechanism to achieve an efficient termination detection algorithm as follows.

The idea is that each thread is associated with a mutex whose status corresponds to the status of the thread: whenever a thread is idle, its corresponding mutex is unlocked and conversely, whenever the thread is busy, its mutex is locked.

In order to detect termination we run a separate thread performing the termination detection. The termination detection algorithm tries to lock mutexes of all worker threads, one by one, using the lock-or-fail behaviour. If it succeeds in locking all mutexes (all working threads are idle) it proceeds to check the communication queues. If these are empty, the termination has occurred and the algorithm terminates. Pseudo-code for the algorithm is shown in Fig. 4.

In order to cope with the termination detection algorithm, every working thread is augmented so that it locks the corresponding mutex whenever it starts processing pending work, and unlocks the mutex whenever it becomes idle. To reduce overhead caused by repeated polling for incoming work, the thread, in addition, enters a sleeping mode after becoming idle. Since we run the termination detection algorithm in a dedicated thread, the termination detection thread may wake up threads that have pending work, but are sleeping due to

**Fig. 4** Termination detection in shared-memory

```
Require: $threads$: array of Thread, Thread contains $idleMutex$ and $idleCondition$, $fifo$
Ensure: termination has occurred iff true is returned
  $mutex$: Mutex, $cond$: Condition, $held$: array of Boolean
  $busy \leftarrow$ false
  for $t$ in $threads$ do
    if trylock($t.idleMutex$) then
      $held[t] \leftarrow$ true
    else
      $held[t] \leftarrow$ false
      $busy \leftarrow$ true
  for $t$ in $threads$ do
    if not empty( $t.fifo$ ) then
      $busy \leftarrow$ true
      if $held[t]$ then
        signal($t.idleCondition$)
  for $t$ in $threads$ do
    unlock( $t.idleMutex$ )
  return not $busy$
```

previous idling; i.e., if the termination thread has successfully grabbed any locks and some queues belonging to those locked threads are found non-empty, the corresponding threads are awakened. After every run of the termination detection algorithm, all grabbed locks are released again.

Moreover, although this algorithm works correctly as-is, it is rather inefficient if the termination detection thread is left running in a loop. Therefore, the termination detection thread goes to sleep after every iteration and is woken up by any worker thread that goes idle.

This modification introduces a race-condition to the algorithm. If the last thread going to sleep wakes up the termination detection thread, which then runs the algorithm before the calling thread manages to go to sleep, the system may deadlock. We solved the problem with further technical modifications of the algorithm; however, we do not list these modification from simplicity reasons.

An alternative approach would be to synchronously execute the termination detection algorithm in the thread that has become idle; but due to the nature of the system, the above procedure is more practical code-wise and only incurs very insignificant overhead.

## 3.6 Workload partitioning

One of the traditional approaches, when exploring the state space of an implicitly specified model, is that the algorithm starts from the initial state and using a transition function, generates successors of every explored state. Visited states are stored in a hash table, to facilitate quick insertion of newly visited states and quick lookup of states that have already been visited.

The usual approach in distributed algorithms is to partition the state space statically, using a partition function [17,19] (which is usually in turn based on a hash function over the state representation). This partition function unambiguously assigns each state to one of the computation nodes. Same approach can be leveraged in shared memory computation, where each thread of control assumes ownership of a private hash table and potentially also a private memory area for storing actual state representations.

All the processors (and therefore threads of control) share a single continuous block of local memory, with uniform accessibility from all the CPUs and/or cores. This gives us two new options, compared with the situation in distributed environment. First, if several hash tables are used, threads can look into tables they do not own, and the second option is to have a single shared hash table, used by all the threads.

*Static and Dynamic Partitioning.* In our research on this topic in [8], we have arrived to the conclusion that when available (i.e. the algorithm allows), static partitioning is preferable to dynamic. Such a scheme leads to distinct hash tables for each thread and appears to improve both performance and scalability by a significant margin. The issues with shared hash table are a subject of further research, and unfortunately, we cannot give more insight into the cause of these problems at this time.

Therefore, we have opted for statically partitioning the state space and using private hash tables. The implementation used in this paper is fully based on this approach. For an experimental evaluation of the shared hash table options, please refer to [8].

## 3.7 Implementing OWCTY in shared-memory

As can be seen from the pseudo-code (refer to Fig. 5), the main OWCTY loop consists of few steps, namely, reachability, elimination and reset. All of them can be parallelised, but only on their own, which requires a barrier after each of them.

The algorithm uses a BFS state space visitor to implement both reachability and elimination. The underlying BFS is currently implemented using a partition function, i.e., every state is unambiguously assigned to one of the threads. The

```
Require: $initial$ is initial state
  $S \leftarrow$Reachability$(initial)$
  $old \leftarrow \emptyset$
  while $S \neq old$ do
    $old \leftarrow S$
    $S \leftarrow$Reset$(S)$
    $S \leftarrow$Reachability$(S)$
    $S \leftarrow$Elimination$(S)$
  return $S \neq \emptyset$
```

**Fig. 5** OWCTY pseudo-code. The reset phase (re-)initialises internal per-vertex bookkeeping for the remaining two phases of the algorithm

framework in which the algorithm is implemented offers a multi-threaded BFS implementation based on static state space partitioning. The algorithm itself is only presented with resulting transition and node expansion events, unconcerned with the partitioning or communication details.

The barriers (i.e. barrier synchronisation) are straightforwardly implemented using the termination detection algorithm presented—the computation is initiated by the main thread, and the termination detection is then executed in this same thread, which also doubles as a scheduler. When the step terminates, the main thread prepares the next step, spawns the worker threads and initiates the computation again. Since the hash table is always thread-private, i.e. owned exclusively by a single thread, the main thread has to transfer the hash table among different threads in the serial portion of computation. This is nonetheless done cheaply (few pointer operations only) so is likely not worth parallelising.

## 4 Experiments

### 4.1 Methodology

The main testing machine we have used is a 16-way AMD Opteron 885 (8 CPU units with 2 cores each). A second testing configuration has been a 4-way Intel Xeon 5130. All timed programs were compiled using unpatched gcc 4.2.2, using -O2. We have used both 32- and 64-bit builds, using -m32 and -m64, respectively. If not specified otherwise, a 64-bit build has been used, due to memory demands of the verification runs (32-bit pointers can only address up to 4 gigabytes of memory, part of which is reserved by the operating system). The 32-bit version has only been used for comparison with few of the smaller models.

For this paper, our main concern is speed and scalability; therefore, we focus on these two parameters. Measurement has been done using standard UNIX `time` command, which measures real and cpu times used by a program.

For the experimental evaluation we implemented algorithms upon the state generator from DiVinE [7]. All the models we have used are listed in Table 1 including the verified properties. The models come from the BEEM database [36] that contains the models in DiVinE-native modelling language as well as in ProMeLa. We used ProMeLa models for comparison with the SPIN model checker.

### 4.2 Generator influence

In addition to these models, we have implemented a special-purpose ("dummy") state space generator which generates very small states very quickly, to evaluate scalability when using a high-performance generator. This is important to identify the role of our current generator, which is known to be sub-optimal in the performance and scalability of the framework. The results are presented in Fig. 6. It can be seen that, although the scaling is roughly 1:2 (number of cores needs to be quadrupled to double the speed, ie efficiency is about 50%), it is fairly flat across the board (i.e. the speedup between 1 and 2 cores is not much higher than speedup between 2 and 4 cores and so on). This is an interesting result, since it gives us a good lower bound estimate on scaling behaviour of the system when we improve the state space generator. Scalability is inversely proportional to communication overhead—a faster generator means the proportion of time spent in communication is higher than with a slower one.

In addition to the "dummy" generator, a model from BEEM with similar runtime on a single core is plotted in the same figure, for comparison. The expected scalability behaviour with improved state space generator falls between that of these two (i.e. worse than current behaviour with realistic models, but better than that of the "dummy" generator).

### 4.3 Results

We report both runtimes and speedup for BFS reachability in Fig. 7. Measurements for the implementation of OWCTY algorithm (for full LTL model checking) are provided in Fig. 8. These were obtained using a 64-bit build on the 16-way AMD Opteron machine. Another set of data points comes from the 4-way Intel machine (a 64-bit build again), visualised in Fig. 9 for reachability.

Some of the phenomena visible in the plots need to be explained. First of all, it needs to be noted that the exact distribution of graph vertices among worker threads strongly depends on actual number of worker threads used, due to distribution scheme used (please refer to Sect. 3.6: DiVinE Multi-Core uses hash$(vertex)/threadcount$ to assign a given vertex to its owner thread). We call an edge a "cross" edge when the two vertices it connects belong to different threads. Clearly, communication overhead depends on proportion of such "cross" edges in the system. Moreover, this proportion depends on exact partitioning of the state space, and this varies with the actual number of threads used. This sometimes leads to situations where adding more CPU cores

**Table 1** Model and property descriptions

| Acronym | Description | Property (LTL formula) |
|---|---|---|
| *anderson* | Anderson's queue lock mutual exclusion algorithm `anderson.6.dve` | If $P_0$ waits for $CS$ then it will eventually get there. $G(wait0 \implies F(cs0))$ `anderson.6.prop2.dve` |
| *at* | Discrete time model of Alur-Taubenfeld fast timing-based mutual exclusion algorithm. `at.5.dve` | $N/A$ |
| *elevator* | An elevator controller model with 3 floors. `elevator2.3.dve` | Cab passes level 0 at most once without serving it, after it has been requested $G(r0 \implies (\neg l_0 U(l_0 U(\neg l_0 U(l_0 U(l_0 \wedge open))))))$ `elevator2.3.prop4.dve` |
| *leader* | The algorithm operates on a ring of N processes. Each process is assigned a unique number. The purpose of this algorithm is to find the largest number assigned to a process. `leader_election.6.dve` | Eventually a leader will be elected $F(elected)$ `leader_election.6.prop2.dve` |
| *leader_s* | A scaled-down version of *leader* that fits the memory available to 32-bit builds. `leader_election.5.dve` | Eventually a leader will be elected. $F(elected)$ `leader_election.5.prop2.dve` |
| *telephony* | Model of a telecommunication service with some features (call forward when busy, ring back when free). `telephony.7.dve` | $N/A$ |
| *peterson* | Peterson's mutual exclusion protocol for N processes. ($N = 4$) `peterson.4.dve` | Someone is in critical section infinitely many times. $G(F(SomeoneInCS))$ `peterson.4.prop4.dve` |

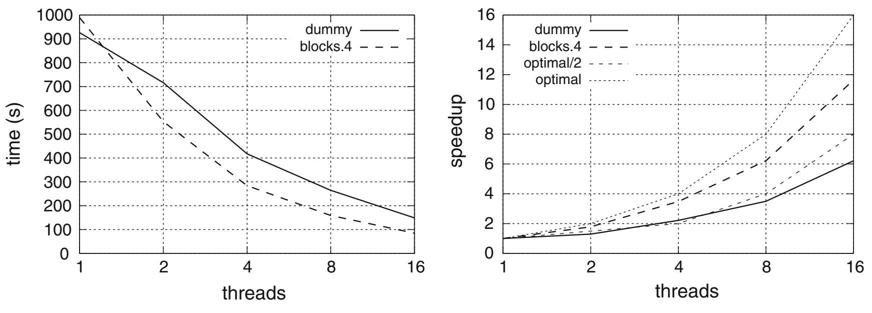


**Fig. 6** Timing and speedup of reachability analysis with high-performance state space generator (designated "dummy" in the plot). The state space has 256 000 005 states and 1 536 000 031 transitions. The `blocks.4.dve` model from `BEEM` is given for comparison

to the computation changes the partitioning of the graph in such a way that the higher communication overhead cancels out the benefit of increased computational power (and sometimes even causes the whole computation to slow down).

Moreover, it can be seen that the OWCTY algorithm, which has inherently higher communication costs than simple reachability, has accordingly poorer scalability behaviour. Another issue that contributes to the inferior scalability of OWCTY is the ordering restriction imposed in the "elimination" pass, which reduces the amount of work that can be done in parallel and therefore impedes obtained speedup.

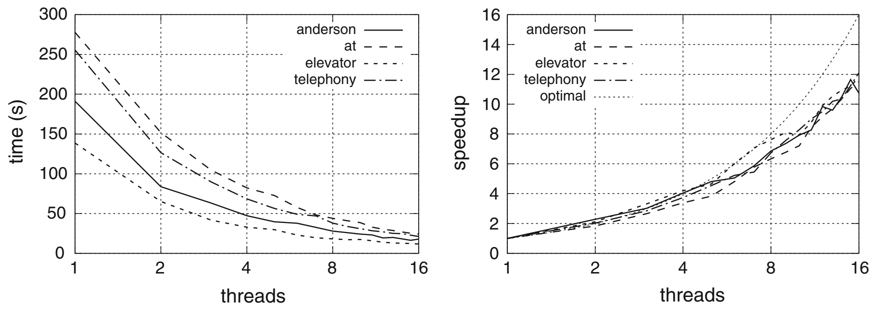


**Fig. 7** Timing and speedup of reachability analysis

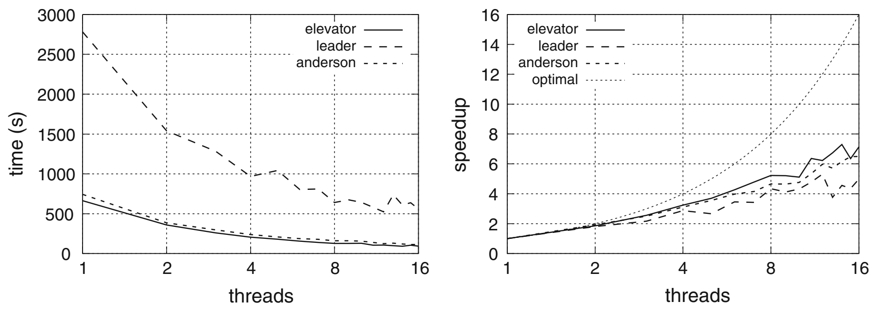


**Fig. 8** Timing and speedup of LTL model checking, the algorithm used is OWCTY

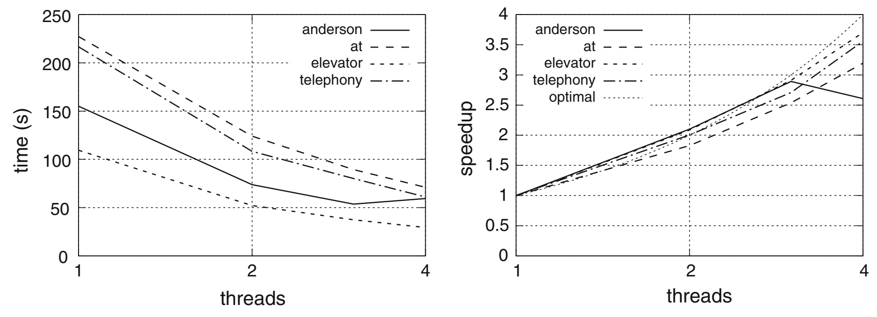


**Fig. 9** Timing and speedup of reachability analysis on a different platform (4-way Intel Xeon with 16G RAM)

In addition to these experiments, we have performed a few comparison runs, first for 32/64 bit builds, with results presented in Figs. 10 and 11. These have been done in order to establish the effect of pointer width on performance and scalability. The conclusion we can draw from the experiment is that while the wider pointers incur a performance penalty on a single core run (which is observable in reachability, but negligible in OWCTY), this penalty is eventually evened out as cores are added (which in turn means that the penalty is divided evenly among the cores).

Furthermore, the effect of custom-made pool allocator is shown in Fig. 12. It can be seen that the tailored allocator, indeed, helps with scalability on the extreme right end of the

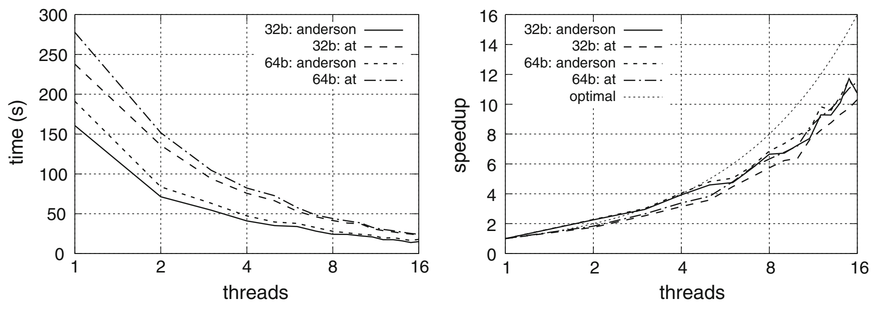


**Fig. 10** Comparison of 32 and 64 bit builds of reachability in regard to both absolute running time and achieved speedup

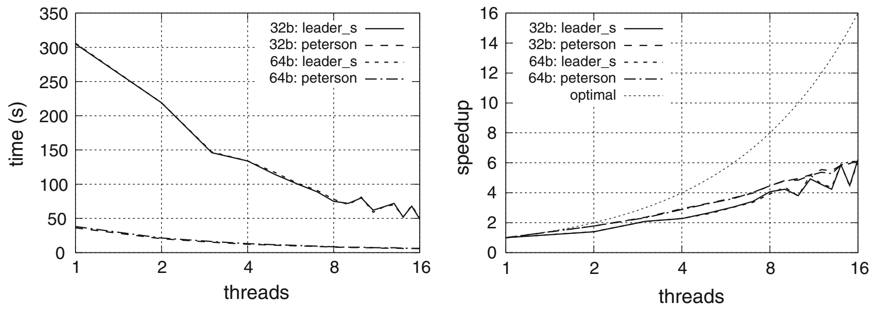


**Fig. 11** Comparison of 32 and 64 bit builds of OWCTY, in regard to both absolute running time and achieved speedup

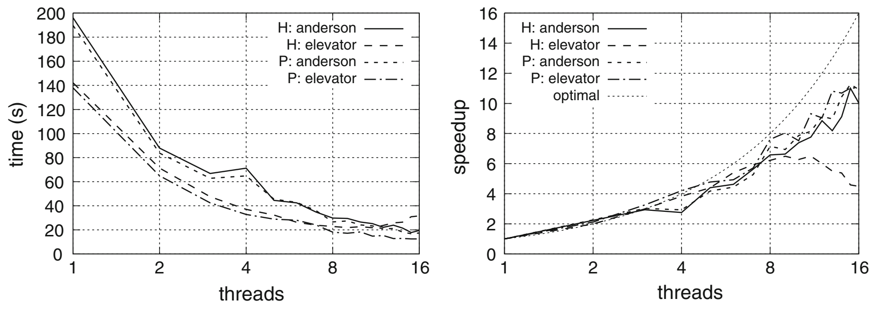


**Fig. 12** Impact of custom, fine-tuned allocator on reachability. The runs with all allocation driven by HOARD are designated H, the ones using a custom pool allocator are designated with P

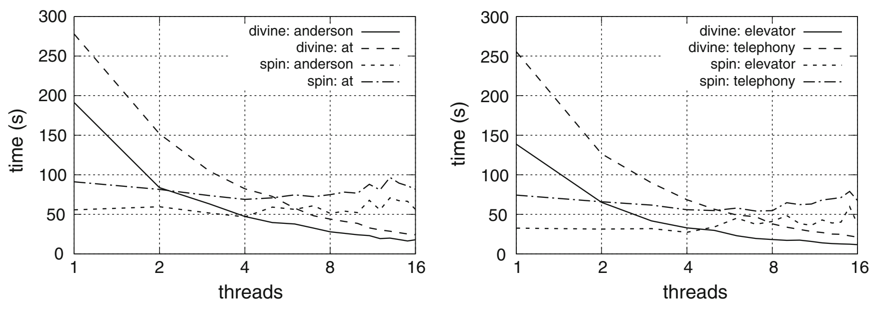


**Fig. 13** Timing of DiVinE and SPIN, reachability analysis

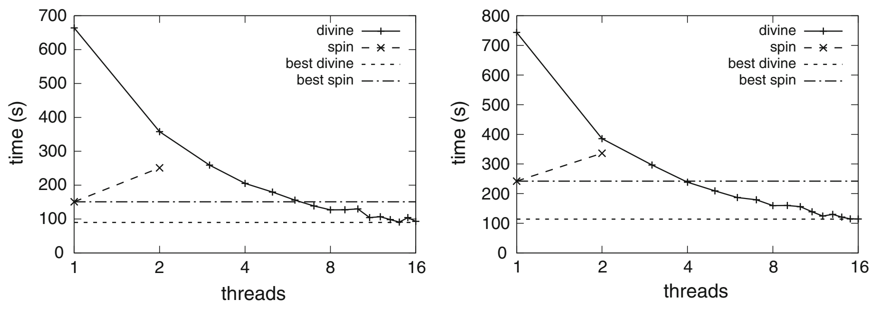


**Fig. 14** Timing of DiVinE and SPIN, LTL model checking. The SPIN algorithm is limited to 2 cores. The model on the *left* is elevator. On *the right*, anderson with $GF(someoneInCS)$ as a property

plot (between 8 and 16 cores)—more so with some models than with others.

### 4.4 Comparison with SPIN version 5.1

SPIN-generated verifier has been used with parameters `-E -A -w27 -m5000000` and compiled with `-DMEMLIM = 8000 -DNOREDUCE -DVMAX = 512`, plus `-DSAFETY` for reachability. We have used bigger stack or hash table than strictly necessary (to avoid running into excessive hash table collisions or exceeding search depth; determining the best sizes for each model would be slightly impractical). For the NCORE $>$ 1 runs, it has also been necessary to increase the system-shared memory limit. For the Anderson model, the stack limit needed to be quadrupled to facilitate verification.

**Table 2** Scalability behaviour of DiVinE Multi- Core reachability on a wide selection of BEEM models

| BEEM model | 16-way AMD Opteron, 64G RAM | | | | | 4-way Intel Xeon, 16G RAM | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 1 | 2 | 3 | 4 |
| anderson.6 | 2:29 | 1:22 | 1:05 | 27 | 16 | 2:11 | 1:11 | 51 | 55 |
| at.5 | 4:22 | 2:28 | 1:20 | 42 | 22 | 3:31 | 1:59 | 1:26 | 1:08 |
| at.6 | 25:40 | 14:08 | 7:55 | 4:12 | 2:16 | 20:04 | 11:29 | 8:44 | 6:42 |
| bakery.6 | 1:35 | 44 | 23 | 14 | 8 | 1:16 | 37 | 26 | 20 |
| blocks.4 | 16:29 | 9:13 | 4:44 | 2:39 | 1:25 | 13:05 | 7:15 | 4:56 | 4:01 |
| elevator.5 | 30:39 | 24:29 | 18:22 | 19:17 | 18:46 | 24:17 X | 18:40 X | 14:20 X | 14:08 X |
| elevator2.3 | 1:42 | 57 | 31 | 17 | 11 | 1:23 | 46 | 35 | 28 |
| elevator_planning.2 | 2:17 | 1:08 | 33 | 16 | 9 | 2:02 | 1:01 | 42 | 31 |
| firewire_link.3 | 1:24:11 | 40:25 | 21:58 | 12:07 | 6:56 | 30:49 X | 14:27 X | 10:48 X | 7:41 X |
| firewire_link.5 | 3:39 | 2:20 | 1:29 | 42 | 29 | 3:18 | 2:13 | 1:55 | 1:17 |
| fischer.6 | 1:19 | 44 | 23 | 12 | 7 | 1:04 | 35 | 25 | 19 |
| frogs.4 | 4:43 | 1:58 | 1:00 | 35 | 17 | 3:14 | 1:23 | 58 | 47 |
| frogs.5 | 51:13 | 19:59 | 10:19 | 5:15 | 3:06 | 35:03 | 15:06 | 11:03 | 8:14 |
| lamport.7 | 5:16 | 2:50 | 1:28 | 47 | 26 | 4:17 | 2:17 | 1:51 | 1:17 |
| lamport.8 | 9:07 | 4:51 | 2:30 | 1:23 | 43 | 7:23 | 4:04 | 2:44 | 2:12 |
| lamport_nonatomic.5 | 21:53 | 12:51 | 7:12 | 4:33 | 3:22 | 19:02 | 10:47 | 11:40 | 6:54 |
| lann.6 | 23:57 | 12:00 | 6:20 | 3:20 | 1:48 | 20:34 | 10:17 | 7:05 | 5:34 |
| lann.7 | 35:50 | 18:18 | 9:38 | 5:16 | 2:46 | 29:04 | 14:45 | 10:37 | 10:48 |
| leader_filters.6 | 25:03 | 12:40 | 5:53 | 3:09 | 1:43 | 17:40 X | 8:31 X | 6:53 | 4:24 X |
| leader_filters.7 | 3:04 | 1:35 | 1:05 | 36 | 19 | 2:33 | 1:26 | 1:07 | 59 |
| loyd.3 | 1:04:03 | 30:03 | 13:49 | 7:20 | 3:33 | 41:42 | 11:37 X | 5:38 X | 6:47 X |
| mcs.5 | 8:56 | 5:25 | 2:39 | 1:26 | 47 | 7:23 | 3:57 | 2:48 | 2:19 |
| msmie.4 | 1:15 | 35 | 19 | 11 | 6 | 1:05 | 32 | 21 | 18 |
| peg_solitaire.2 | 2:57:57 | 1:27:13 | 51:23 | 27:32 | 18:53 | 2:15:36 | 33:58 X | 22:32 X | 15:52 X |
| peterson.5 | 23:03 | 11:58 | 6:19 | 3:26 | 1:52 | 18:14 | 9:21 | 6:40 | 5:19 |
| peterson.6 | 33:09 | 16:25 | 8:38 | 4:43 | 2:39 | 25:05 | 12:40 | 9:25 | 7:14 |
| peterson.7 | 24:00 | 11:38 | 6:14 | 3:11 | 1:48 | 19:06 | 9:22 | 6:41 | 5:21 |
| phils.6 | 3:56 | 2:20 | 1:15 | 43 | 31 | 3:18 | 1:58 | 1:23 | 1:10 |
| phils.7 | 27:17 | 15:37 | 8:25 | 4:15 | 2:29 | 21:58 | 8:12 X | 4:29 X | 3:03 X |
| phils.8 | 14:10 | 7:55 | 4:13 | 2:28 | 1:41 | 11:04 | 4:40 X | 2:25 X | 1:41 X |
| production_cell.6 | 1:59 | 1:02 | 33 | 18 | 11 | 1:48 | 56 | 38 | 31 |
| schedule_world.3 | 1:24:15 | 46:07 | 24:20 | 14:53 | 8:56 | 1:09:28 | 11:31 X | 7:01 X | 4:43 X |
| sokoban.3 | 13:55 | 12:22 | 12:24 | 12:24 | 13:42 | 10:09 X | 10:16 X | 10:09 X | 10:15 X |
| szymanski.5 | 13:05 | 6:19 | 4:44 | 3:08 | 2:23 | 11:08 | 5:35 | 6:11 | 4:21 |
| telephony.4 | 2:11 | 1:11 | 37 | 20 | 11 | 1:50 | 59 | 43 | 33 |
| telephony.5 | 1:53:48 | 43:39 X | 32:26 | 17:33 | 9:20 | 40:07 X | 6:18 X | 3:41 X | 3:34 X |
| telephony.7 | 3:56 | 2:04 | 1:06 | 37 | 20 | 3:23 | 1:45 | 1:18 | 1:00 |

Only models with a single-core runtime longer than 1 min are shown. The X mark means that the run did not finish due to memory exhaustion

We have compared both dual-core LTL model checking and multi-core reachability analysis. The results are presented in a time-to-cores plot in Fig. 13 for reachability, and in Fig. 14 for LTL model checking.

It is easily seen that for a single-core run, SPIN is much faster on both LTL and reachability, thanks to its faster state-space generator. However, with increasing number of cores, the superior scalability of DiVinE Multi- Core eventually closes the gap. We attribute this to problems in SPIN's implementation of parallel reachability that prevent it from scaling on communication-intensive workloads. In [29], it has been demonstrated that the SPIN implementation is capable of

**Table 3** Scalability behaviour of DiVinE Multi- Core liveness checking on a wide selection of BEEM models

| BEEM model | 16-way AMD Opteron, 64G RAM | | | | | 4-way Intel Xeon, 16G RAM | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 1 | 2 | 3 | 4 |
| anderson.6.prop2 | 8:40 | 4:57 | 3:30 | 2:10 | 1:30 | 7:14 | 4:05 | 3:07 | 2:52 |
| anderson.6.prop4 | 11:41 | 6:27 | 4:46 | 2:45 | 1:56 | 9:39 | 5:23 | 4:10 | 3:56 |
| bopdp.3.prop1 | 11 | 6 | 4 | 3 | 3 | 9 | 5 | 4 | 4 |
| bopdp.3.prop3 | 15 | 8 | 6 | 4 | 3 | 13 | 7 | 6 | 5 |
| bopdp.3.prop4 | 13 | 8 | 5 | 4 | 3 | 11 | 7 | 5 | 4 |
| elevator.5.prop3 | 49:21 | 38:16 | 29:56 | 30:45 | 32:04 | 16:35 X | 10:44 X | 7:05 X | 7:15 X |
| elevator2.3.prop4 | 9:27 | 5:40 | 3:28 | 2:07 | 1:35 | 8:18 | 4:51 | 3:50 | 3:05 |
| elevator2.3a.prop4 | 1:29 | 56 | 35 | 21 | 14 | 1:18 | 47 | 34 | 29 |
| lamport.5.prop4 | 28 | 17 | 10 | 7 | 5 | 23 | 14 | 10 | 8 |
| lamport.7.prop4 | 23:12 | 13:23 | 8:10 | 5:16 | 3:39 | 19:41 | 10:58 | 8:28 | 6:49 |
| leader_election.5.prop2 | 4:45 | 3:34 | 2:09 | 1:11 | 47 | 4:17 | – | 2:01 | 1:53 |
| leader_election.6.prop2 | 43:00 | – | 16:16 | 10:49 | 8:28 | 38:36 | 12:08 X | 5:40 X | 6:13 X |
| leader_filters.5.prop2 | 26 | 15 | 9 | 8 | 6 | 22 | 15 | 9 | 8 |
| leader_filters.6.prop2 | 1:06:16 | 37:05 | 21:42 | 15:54 | 11:06 | 12:35 X | 5:03 X | 3:16 X | 2:33 X |
| leader_filters.7.prop2 | 9:04 | 5:10 | 3:08 | 2:25 | 1:45 | 7:34 | 5:02 | 3:23 | 2:53 |
| mcs.3.prop4 | 14 | 8 | 5 | 4 | 3 | 12 | 7 | 6 | 5 |
| mcs.5.prop4 | 36:36 | 22:42 | 13:27 | 8:36 | 6:14 | 31:06 | 10:53 X | 8:56 X | 7:26 X |
| peterson.5.prop4 | 1:38:18 | 52:58 | 32:12 | 20:20 | 13:35 | 26:06 X | 4:27 X | 3:09 X | 2:45 X |
| peterson.7.prop4 | 1:51:46 | 58:48 | 36:18 | 22:40 | 15:41 | 26:22 X | 7:49 X | 5:55 X | 4:19 X |
| public_subscribe.4.prop1 | 19 | 10 | 7 | 5 | 4 | 17 | 9 | 8 | 6 |
| rether.5.prop5 | 1:27 | 1:00 | 32 | 22 | 20 | 1:15 | 58 | 39 | 28 |
| rether.6.prop2 | 50 | 33 | 24 | 22 | 14 | 42 | 28 | 22 | 21 |
| rether.7.prop2 | 40 | 20 | 19 | 11 | 9 | 34 | 18 | 18 | 17 |
| rether.7.prop5 | 2:22 | 1:12 | 58 | 38 | 27 | 2:06 | 1:05 | 58 | 55 |
| synapse.6.prop3 | 8 | 5 | 3 | 2 | 2 | 7 | 4 | 3 | 3 |
| szymanski.4.prop4 | 1:20 | 51 | 39 | 33 | 28 | 1:09 | 43 | 39 | 35 |

Only valid model/property combinations are included. The X mark means that the run did not finish due to memory exhaustion

scaling whenever successor generation is very expensive (and therefore, the communication overhead is much smaller). We were, however, unable to reproduce this scalability in our models, where successor generation is relatively cheap. Unfortunately, we cannot use the models used in [29], as these are only available in ProMeLa, if at all. Moreover, in the BEEM database, only few models are available where the DVE and ProMeLa state spaces match exactly. Nevertheless, we believe that the models used are diverse enough to be representative of the general trend.

We have also executed a batch of SPIN-only experiments (using version 5.1), but it has been difficult to extract comparisons of single- and multi- core runs. The multi-core version often fails to report errors correctly (for example, when the single-core run reports an out of memory error, the 4-core run does not, but visits smaller number of states, which probably means it also ran out of memory without reporting the fact). Where we were able to reliably measure reachability runtime, the improvement using 4 cores over a single core fell between factor 1 and 2 (we were unable to find a model in BEEM where the speedup would exceed 2 when using 4 cores). (Tables 2, 3.)

## 5 Conclusions

We observe that the algorithms employed by *DiVinE Multi-Core* scale fairly well on multiple cores. Our current OWCTY-based, multi-threaded implementation is able to reduce runtime with increasing number of cores to up to 12 cores, and for some models, even to 16 cores, which is a definite improvement over the MPI implementation of the algorithm if run on the same shared memory platform.

This fulfils our goal of implementing a scalable multi-core LTL model checker. It maintains a linear time complexity for majority of LTL properties verified in practice and provides scalability that makes it practical to use on machines with several CPU cores available. Moreover, in the range of 4–8 cores, the tool performance rivals that of SPIN and even exceeds it with higher core numbers, which is in itself a considerable achievement.

From the profiling work we have done, it is clear that the current most important bottleneck of DiVinE is its state generator. The experimental data presented in the paper support this observation, especially when compared with SPIN, with its famously fast state space generator. Improvements in this area should reduce the absolute running times, while it may negatively affect relative scalability. Nevertheless, the current implementation of algorithm and supporting mechanisms still offer a speedup close to (number-of-cores/2) even when used with a very fast state space generator.

To sum up, this paper presents a significant achievement in the stated goal of implementing scalable algorithms and support code for reachability and LTL model checking in *DiVinE Multi-Core*.

## References

1. Barnat, J., Brim, L., Chaloupka, J.: Parallel breadth-first search LTL model-checking. In: IEEE International Conference on Automated Software Engineering (ASE'03), pp. 106–115. IEEE Computer Society Press (2003)
2. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core LTL model-checkin. In: Model Checking Software (SPIN'07), volume 4595 of LNCS, pp. 187–203. Springer (2007)
3. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core—a parallel LTL model-checker. In: Automated Technology for Verification and Analysis (ATVA'08), volume 5311 of LNCS, pp. 234–239. Springer (2008)
4. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Model Checking Software (SPIN'01), volume 2057 of LNCS, pp. 200–216. Springer (2001)
5. Barnat, J., Brim, L., Černá, I.: Property driven distribution of nested DFS. In: International Workshop on Verification and Computational Logic (VCL'02), pp. 1–10. University of Southampton, UK. Technical Report DSSE-TR-2002-5 in DSSE (2002)
6. Barnat, J., Brim, L., Černá, I.: Cluster-based LTL model checking of large systems. In: Formal Methods for Components and Objects (FMCO'05), number 4111 in LNCS, pp. 259–279. Springer (2006)
7. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE—a tool for distributed verification (Tool Paper). In: Computer Aided Verification (CAV'06), volume 4144 of LNCS, pp. 278–281. Springer (2006)
8. Barnat, J., Ročkai, P.: Shared hash tables in parallel model checking. In: Parallel and Distributed Methods in verification (PDMC'07), pp. 81–95. CTIT, University of Twente (2007)
9. Behrmann, G., Hune, T.S., Vaandrager, F.W.: Distributed timed model checking—How the search order matters. In: Computer Aided Verification (CAV'00), volume 1855 of LNCS, pp. 216–231. Springer (2000)
10. Bell, A., Haverkort, B.R.: Sequential and distributed model checking of Petri Net specifications. Int. J. Softw. Tools Technol. Transfer **7**(1), 43–60 (2005)
11. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: a scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pp. 117–128. ACM Press (2000)
12. Blom, S., Orzan, S.: A distributed algorithm for strong bisimulation reduction of state spaces. Int. J. Softw. Tools Technol. Transfer **7**(1), 74–86 (2005)
13. Bollig, B., Leucker, M., Weber, M.: Parallel model checking for the alternation free $\mu$-calculus. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of LNCS, pp. 543–558. Springer (2001)
14. Brim, L., Černá, I., Moravec, P., Šimša, J.: 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, pp. 352–366. Springer (2004)
15. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL model checking based on negative cycle detection. In: Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01), volume 2245 of LNCS, pp. 96–107. Springer (2001)
16. Brim, L., Černá, I., Moravec, P., Šimša, J.: How to order vertices for distributed LTL model-checking based on accepting predecessors. ENTCS **132**(2), 3–18 (2006)
17. Caselli, S., Conte, G., Marenzoni, P.: Parallel state space exploration for GSPN models. In: Applications and Theory of Petri Nets (PN'95), volume 935 of LNCS, pp. 181–200. Springer (1995)
18. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection (set based approach). In: Model Checking Software (SPIN'03), volume 2648 of LNCS, pp. 49–73. Springer (2003)
19. Ciardo, G., Gluckman, J., Nicol, D.M.: Distributed state space generation of discrete-state +stochastic models. INFORMS J. Comput. **10**(1), 82–93 (1998)
20. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst. Des. **1**, 275–288 (1992)
21. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of LNCS, pp. 420–434. Springer (2001)
22. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: Model Checking Software (SPIN'01), volume 2057 of LNCS, pp. 217–234. Springer (2001)
23. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. **345**(1), 60–82 (2005)
24. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. **345**(1), 60–82 (2005)
25. Grumberg, O., Heyman, T., Ifergan, N., Schuster, A.: achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In: Correct Hardware Design and Verification Methods (CHARME'05), volume 3725 of LNCS, pp. 129–145. Springer (2005)
26. Grumberg, O., Heyman, T., Schuster, A.: Distributed model checking for $\mu$-calculus. In: Computer Aided Verification (CAV'01), volume 2102 of LNCS, pp. 350–362. Springer (2001)
27. Haverkort, B.R., Bell, A., Bohnenkamp, H.C.: On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri Nets. In: International Workshop on Petri Net and Performance Models (PNPM'99), pp. 12–21. IEEE Computer Society Press (1999)
28. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)

29. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the SPIN model checker. IEEE Trans. Softw. Eng. **33**(10), 659–674 (2007)
30. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth first search. In: The SPIN Verification System, pp. 23–32. American Mathematical Society (1996)
31. Inggs, C., Barringer, H.: CTL* model checking on a shared memory architecture. Formal Methods Syst. Des. **29**(2), 135–155 (2006)
32. Lafuente, A.L.: Simplified distributed LTL model checking by localizing cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, July 2002
33. Leung, H.-F., Ting, H.-F.: An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. IEEE Trans. Parallel Distrib. Syst. **8**(5), 538–543 (1997)
34. Michael, M.M.: Scalable lock-free dynamic memory allocation. SIGPLAN Not. **39**(6), 35–46 (2004)
35. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Symposium on Principles of Distributed Computing (PODC'96), pp. 267–275. ACM Press (1996)
36. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Model Checking Software (SPIN'07), volume 4595 of LNCS, pp. 263–267. Springer (2007)
37. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), volume 3440 of LNCS, pp. 174–190. Springer (2005)
38. Talbot, S.: Performance tuning of programs for shared-memory multiprocessors. Master's thesis, Department of Computing, Imperial College, London (1995)
39. Tarjan, R.: Depth first search and linear graph algorithms. SIAM J. Comput. **2**, 146–160 (1972)
40. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Verification, Model Checking, and Abstract Interpretation (VMCAI'07), volume 4349 of LNCS, pp. 137–150. Springer (2007)
41. Verstoep, K., Bal, H., Barnat, J., Brim, L.: Efficient large-scale model checking. In: 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009). IEEE (2009)

# Efficient Large-Scale Model Checking*

Kees Verstoep, Henri E. Bal
Dept. of Computer Science, Fac. of Sciences
VU University, Amsterdam, The Netherlands
{versto,bal}@cs.vu.nl

Jiří Barnat, Luboš Brim
Dept. of Computer Science, Fac. of Informatics
Masaryk University, Brno, Czech Republic
{barnat,brim}@fi.muni.cz

## Abstract

*Model checking is a popular technique to systematically and automatically verify system properties. Unfortunately, the well-known state explosion problem often limits the extent to which it can be applied to realistic specifications, due to the huge resulting memory requirements. Distributed-memory model checkers exist, but have thus far only been evaluated on small-scale clusters, with mixed results. We examine one well-known distributed model checker, DiVinE, in detail, and show how a number of additional optimizations in its runtime system enable it to efficiently check very demanding problem instances on a large-scale, multi-core compute cluster. We analyze the impact of the distributed algorithms employed, the problem instance characteristics and network overhead. Finally, we show that the model checker can even obtain good performance in a high-bandwidth computational grid environment.*

## 1 Introduction

One of the main challenges in the field of computer science is to provide formalisms, techniques, and efficient tools for assessing the correctness or other functional properties of increasingly complex computer systems. One such technique is model checking, which systematically (and automatically) checks whether a model of a given system satisfies a desired property. This automated technique for verification and debugging has developed into a mature and widely used approach.

Conventional sequential model checking techniques have high memory requirements and are very computationally intensive; they are thus unsuitable for handling real-world systems that exhibit complex behaviors which cannot be captured by simple models having a small or regular state space. Various authors have proposed ways of solving this problem by either using powerful shared-memory multiprocessors (e.g., multi-core machines) or by distributing the memory requirements over several machines (e.g., on a cluster of workstations).

Memory requirements are often the bottleneck in being able to solve a problem at all. Therefore, it can still be beneficial to use algorithms with a slightly higher computational complexity, provided they can be distributed effectively using a large distributed memory. A prominent example in this category is the DiVinE [1] system , which we will focus on in this paper. As DiVinE is especially targeted on model specifications that induce very large state spaces, an important question is to what extent it scales to a large number of compute nodes. Previous research has shown that the different distributed algorithms included in DiVinE can have widely diverse execution times, depending on the model characteristics [1]. We will closely examine two different algorithms that previously were shown to have the best overall performance, and we will analyze their behavior on model instances that require significantly more memory than the ones tackled before.

Models with large search spaces arise naturally from a straightforward formalization of a system under development. To make complete (finite) analysis of such models possible, often simplifying assumptions have to be introduced, with the unfortunate risk of certain inconsistencies escaping analysis. Typically, also, models have to be made amenable for analysis by putting an artificial boundary on the number of resources or processes involved. By scaling the model up from small instances to more realistic proportions, gradually more trust can be gained in the verification results. However, seemingly simple, restricted specifications can still give rise to unexpectedly huge search spaces, also known as the *state explosion* problem. Although abstraction techniques exist which restrict models to their essential core (without losing behavioral characteristics that do require checking), large-scale analysis is often still a necessity in practical cases. For example, the checking of routing protocols for mobile ad hoc networks [29] resulted in verification of various scenarios, several of which could

*This work has been supported in part by the Czech Science Foundation, grants No. 201/09/1389 and 201/09/P497.

not be verified using the efficient (sequential) SPIN [16] model checker, due to their very large state spaces. As DIVINE also supports SPIN specifications [28], additional scenarios can now be verified using a cluster.

The contributions of this paper are as follows. We describe and analyze several optimizations for the DIVINE framework and two of its algorithms that together improve their performance up to 50%. We show that these optimizations allow the algorithms to scale well, up to at least 256 cores, and that they can efficiently exploit modern multicore architectures. We compare the performance of both parallel algorithms on seven representative models having different characteristics, all exhibiting state spaces that are much larger than could be tackled before. We analyze the sensitivity of the algorithms to protocol overhead of the network used, as this can typically have a large impact on parallel performance. Finally, we show that DIVINE, which is largely implemented using asynchronous communication, can now even be run efficiently on a large-scale optical computational grid, despite the much higher (wide-area) latencies on such a platform.

The paper is structured as follows. In Section 2 we examine DIVINE and two of its main parallel algorithms, and we discuss their communication patterns. Section 3 discusses the optimizations we applied to DIVINE, and their effectiveness in improving the performance of both algorithms. Next, Section 4 contains a performance analysis of the optimized model checker on six additional realistic problems with search spaces up to 245 GB. In Section 5 we discuss related work and conclude.

## 2 Distributed-Memory Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The check itself is in principle an exhaustive search in the model. The main technical problem in model checking is the *state explosion* which can occur if the system being verified has many components which make transitions in parallel. The size of the constructed model grows exponentially in the size of the system's description.

Much attention has been paid to the development of approaches to battle the state explosion problem. Many techniques, such as abstraction, state compression, partial order reduction, symbolic state representation, etc., are used to reduce the size of the model, thus allowing a single computer to still process large systems. However, despite impressive progress on these reduction techniques, the memory required to handle large industrial models still exceeds the capacities offered by a single contemporary computer.

One possible approach is to increase the computational power and memory capacity of the system by using a compute cluster, in which the compute nodes communicate via a message passing interface. The use of distributed-memory processing for model checking indeed has gained interest in recent years. Techniques have been developed for both explicit and symbolic model checking, analysis of stochastic and timed systems, equivalence checking and other verification methods.

### 2.1 LTL Model Checking

In this paper we consider one particular model-checking procedure, namely *enumerative LTL model checking*. In LTL model checking, the properties are specified in Linear Temporal Logic, which is a temporal logic suitable to express properties about the future of executions of the system model, e.g., that a condition will eventually be true, or that a condition will be true until another fact becomes true, etc. An efficient procedure to decide LTL model checking problems is based on automata and was introduced by Vardi and Wolper [26]. In this approach, both the model and the LTL formula are associated with an *automaton*, and the LTL model-checking problem is reduced to detecting an *accepting cycle* (i.e., a cycle in which one of the vertices is marked "accepting") in the combined *automaton graph*.

The optimal sequential algorithms for accepting cycle detection use depth-first search (DFS) strategies. The individual algorithms differ in their space requirements, length of the counterexample produced, and other aspects. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best suitable algorithm for enumerative *sequential* LTL model checking. The algorithm was proposed by Courcoubetis et al. [9] and its main idea is to use two interleaved graph searches to detect reachable accepting cycles. The first search discovers accepting states, while the second (the nested one) checks for self-reachability. Another group of optimal algorithms are *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into Strongly Connected Components (SCCs) [25]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counterexamples in general, which can thus be analyzed more conveniently. The time complexity of these algorithms is linear in the size of the graph, i.e., $O(m+n)$, where $m$ is the number of edges and $n$ is the number of vertices.

The effectiveness of the *Nested DFS* algorithm is achieved due to the particular order in which the graph is explored, also guaranteeing that vertices are not re-visited more than twice. In fact, all best-known algorithms rely on the same exploration principle, namely the *postorder* as computed by the DFS. It is a well-known fact that the postorder problem is P-complete and, consequently, a scalable parallel algorithm which would be directly based on DFS postorder is unlikely to exist.

An additional important criterion for a model checking algorithm is whether it works *on-the-fly*. On-the-fly algorithms generate the automaton graph gradually as they explore vertices of the graph. An accepting cycle can thus be detected before the complete set of vertices is generated. On-the-fly algorithms usually assume the graph to be given *implicitly* by the function $F_{init}$ giving the initial vertex and by the function $F_{succ}$ which returns immediate successors of a given vertex.

### 2.2 Parallel Algorithms for LTL Model Checking

In many cases the algorithms as used traditionally are not appropriate to be adapted to parallel architectures. In the case of LTL model checking, all efficient algorithms build on depth-first search exploration of the state space. However, there is no known way to efficiently compute DFS postorder on parallel machines. New algorithms have to be invented to replace the classical ones. We briefly introduce two algorithms for accepting cycle detection that are (among others) implemented in DiVinE. The sequential complexity of these algorithms is worse than for those based on DFS, but both allow solving the LTL model-checking problem on parallel architectures much more efficiently. For a detailed survey on these and other algorithms implemented in DiVinE we refer to [1].

#### OWCTY: Topological Sort Algorithm

The main idea behind the OWCTY (One Way Catch Them Young) algorithm stems from the fact that a directed graph can be topologically sorted if and only if it is acyclic. The core of the cycle detection algorithm is thus an application of the standard linear topological sort algorithm to the input graph. Failure in topologically sorting the graph means the graph contains a cycle. Accepting cycles are detected with multiple rounds (iterations) of the topological sort. Every iteration consists of reachability and elimination procedures. The reachability procedure removes vertices unreachable from an accepting vertex (as these cannot belong to an accepting cycle) and computes indegrees for all remaining vertices. The succeeding elimination procedure recursively eliminates vertices whose predecessor count drops to zero. The algorithm does not work on-the-fly, as the entire automaton graph has to be generated first. Also, the algorithm does not immediately give the accepting cycle; it only checks for its *presence* in the graph. However, the counterexample is easily generated using two additional linear graph traversals, like breadth-first search.

The time complexity of the algorithm is $O(h \cdot m)$ where $h$ is the height of the SCC graph. Here the factor $m$ comes from the computation of the *reachability* and *elimination* functions and the factor $h$ relates to the number of external iterations. In practice, the number of external iterations is very small (up to 40–50), even for very large graphs. This observation is supported by experiments in [11]. Similar results are communicated in [22] where heights of SCC graphs were measured for several models. As reported, 70% of the models have heights smaller than 50.

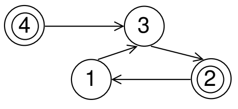


**Figure 1. Undiscovered cycle**

A positive aspect of the algorithm is its extreme effectiveness for *weak automaton graphs*. A graph is weak if in each SCC all the states are accepting or none of them is. For weak graphs only one iteration of the algorithm is necessary to decide about accepting cycles, the algorithm works in linear time and is thus optimal. A study of temporal properties [8] has revealed that verification of up to 90% of LTL properties leads to weak automaton graphs.

#### MAP: Maximal Accepting Predecessors

The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. The algorithm that would be directly derived from this idea requires expensive storing of all proper accepting predecessors for each (accepting) vertex. To remedy this, the algorithm instead stores only a single representative accepting predecessor for each vertex. We presuppose a linear ordering of vertices (given, e.g., by their memory representation) and choose the *maximal accepting predecessor*. For a vertex $u$ we denote its maximal accepting predecessor in the graph $G$ by $map_G(u)$. Clearly, if an accepting vertex is its own maximal accepting predecessor ($map_G(u) = u$), it lies on an accepting cycle. Unfortunately, the opposite does not hold in general. It can happen that the maximal accepting predecessor for an accepting vertex on a cycle does not lie on the cycle. This is exemplified in the graph given in Fig. 1. The accepting cycle $\langle 2,1,3,2 \rangle$ is not revealed due to the greater accepting vertex 4 outside the cycle. However, as vertex 4 does not lie on *any* cycle, it can safely be deleted (marked as non-accepting) from the set of accepting vertices, and the accepting cycle still remains in the resulting graph. This idea is formalized as a *deleting transformation*.

Whenever the deleting transformation is applied to the automaton graph $G$ with $map_G(v) \neq v$ for all $v \in V$, it shrinks the set of accepting vertices by those vertices that do not lie on any cycle. As the set of accepting vertices can change after the deleting transformation has been applied,

```
while (!synchronized()) {
    if ((state = waiting.dequeue()) != NULL) {
        state.work();
        for (tr = state.succs(); tr != NULL; tr = tr.next()) {
            tr.work();
            newstate = tr.target();
            dest = newstate.hash();
            if (dest == this_cpu) waiting.queue(newstate);
            else send_work(dest, newstate);
        }
    }
    else idle();
    process_messages(&waiting);
}
```

**Figure 2. Distributed graph traversal skeleton**

the maximal accepting predecessors must be recomputed. It can happen that even in the graph $del(G)$ the maximal accepting predecessor function is still not sufficient for cycle detection. However, after a finite number of iterations consisting of computing maximal accepting predecessors followed by application of the deleting transformation, an accepting cycle is certified (after which a counterexample can be reconstructed). For an automaton graph without accepting cycles, the application of deleting transformations results in an automaton graph without accepting vertices.

The time complexity of the algorithm is $O(a^2 \cdot m)$, where $a$ is the number of accepting vertices. Here the factor $a \cdot m$ comes from the computation of the *map* function and the factor $a$ relates to the number of iterations. Unlike the OWCTY algorithm, the MAP algorithm does work on-the-fly. Experimental evaluation of this algorithm demonstrated that accepting cycles were typically detected in a very small number of iterations. On the other hand, if there is no accepting cycle in the graph, the number of iterations tends to be very small compared to the size of the graph (up to 40–50). Thus, the algorithm exhibits near linear performance in practice.

## 2.3 DiVinE Tool

The DIVINE toolkit consists of several separate implementations of various LTL model checking algorithms such as described above. Although the algorithms are very different, they follow the same overall pattern, illustrated in Figure 2. All algorithms perform a strict-order independent repeated traversal of a directed graph. Vertices of the graph are very small (typically less than 1 KB), but there are many. To distribute work among compute nodes, the tool partitions the graph into (disjunct) sets of vertices such that each set is owned by one node. This partitioning is implemented using a hash function: every vertex is assigned to a compute node according to the hash value computed from its state representation. Due to the large number of vertices to be distributed, the hash-based partitioning scheme results in a quite well-balanced workload, at the price of minimal locality. The probability that immediate descendants of a vertex belong to the same compute node as the vertex is $1/p$, where $p$ is the number of compute nodes. This means in practice, that significant portions of edges of the graph are so called *cross edges*, i.e., edges whose incident vertices belong to different compute nodes. Basically, every cross edge results in a message to be sent from the compute node owning the source vertex of the edge to the node owning the target vertex of the edge. The message bears information about the explored edge plus a small amount of additional data that is dependent on the algorithm involved. As a result, a huge number of small messages is exchanged among compute nodes during the execution of a DIVINE tool.

Both OWCTY and MAP show a gradually increasing memory usage during their first exploration phase, where the state space is being expanded. During subsequent application phases, memory usage remains constant, as the algorithm-specific state meta-data is preallocated during the first phases. Overall, generating, hashing and comparing states is responsible for a large fraction of the applications' runtime. Furthermore, large-scale graph algorithms like explicit-state model checking have a high data access to computation ratio compared to scientific computing applications [19].

## Distributed Graph Traversal

As shown in Figure 2, the core of each graph traversal algorithm is a while loop over a queue of vertices (states) waiting to be processed. Each time a vertex is dequeued, edges (transitions) emanating from it are enumerated, and for each of them an algorithm-related action is performed. The target vertices are examined, and if they need to be stored locally, they are inserted back into the queue. Non-local vertices are wrapped into messages and sent to their owners. In the serial case the main loop terminates as soon as the queue becomes empty. For distributed algorithms, however, the processing of incoming messages produces new vertices to be inserted into the queue, thus introducing new work. Therefore, the parallel algorithm may terminate only if all local queues are empty and there is no message in transit. To detect this termination condition, Safra/Dijkstra's distributed termination detection algorithm [10] is used; see also [21].

An important observation is that the communication among compute nodes is asynchronous: the algorithms described simply push work to other compute nodes, without triggering replies that require more processing. This aspect also enables an important optimization: work items sent to the same destination can be aggregated into larger messages, significantly reducing the communication overhead. DIVINE implements the communication using asynchronous MPI primitives, which allows for efficient parallel processing on a wide variety of architectures. On the other hand, the use of asynchronous messages may increase the memory demands, both at the application and the communication layer. The more vertices are enqueued in a local queue, the longer it takes before incoming messages are actually received. Since the number of messages is limited by the number of edges, we can observe a shift in the space complexity of the algorithms. Unlike the serial case, where the space complexity of a graph traversal algorithm is asymptotically linear in the number of vertices, the distributed algorithms exhibit space complexity that is asymptotically linear in the number of vertices and edges, hence, up to asymptotically quadratic in the number of vertices. Experimental experience has shown that even for graphs with a relatively small number of transitions (with an average outdegree less than 10), the practical memory demands are significantly increased in the distributed case compared to the serial one, due to incoming message buffering.

To avoid increased memory demands during computation, DIVINE algorithms regularly check for incoming messages. If the content of an incoming message indicates further processing, the appropriate vertex is extracted from the message and it is enqueued to the local queue. If there is no further processing required for the incoming message, the message is discarded immediately.

## 3 Optimizing DIVINE's Performance

The performance of DIVINE was considered to be reasonably good, but had not yet been evaluated on large-scale parallel systems. The original evaluation [1] had investigated performance up to 20 compute nodes, which was the size of the cluster used for the development of DIVINE. For this paper, we were able to make use of the DAS-3 system (discussed below), allowing performance evaluation at a much larger scale. It should be noted that that the applications are far from trivially parallel, as they are very communication intensive, as shown later in this Section. It was soon determined that performance of several DIVINE algorithms did not scale well with a high number of nodes. It was unclear whether this was possibly caused by inherent scalability limitations of the underlying (distributed) algorithms. For example, in the elimination phase of the OWCTY algorithm, the states of the graph must be expanded in topological order. It was previously unclear, whether strictly following this order would force some cores to become idle due to an insufficient amount of work. As our results will show, this is not the case in practice.

An important reason for the initial scalability to drop, was found to be an inefficiency in DIVINE's timer management for its user-level messaging layer. Even though the associated system calls are highly optimized in the Linux kernel, the extent to which they were used still seriously impeded performance when many compute nodes were used. By modifying the timer management to use a cached version of the current time where appropriate (optimization TIMER), large-scale performance was improved significantly.

### 3.1 Optimizations Applied

To investigate possibly remaining performance problems, we started with a bottom-up approach in which DIVINE's networking module (shared by the implementations of parallel algorithms, including MAP and OWCTY) was first instrumented for performance analysis. Every MPI invocation was wrapped in a low-overhead layer that maintained statistics about the call's overhead (e.g., to determine send and receive overhead), and about data transfer rates (both incoming and outgoing). The most important statistics were logged once every three seconds on every CPU core. The combined data was then graphically analyzed to hypothesize causes for the performance degradations, upon which action could be taken. The same approach was recently successfully applied on a distributed application from an entirely different domain (distributed game tree search) that showed a traffic pattern remarkably similar to DIVINE's [27].

DIVINE's receive primitive, called *process_messages* (see Figure 2), was an important target in several of our optimizations. Besides implementing polling, receiving and processing user messages, it is also responsible for timeout-based flushing of pending messages and the handling of distributed termination detection. The following optimizations (with acronyms for reference) were applied:

**Auto-tune receive rate** (RATE) – DIVINE's applications often performed much more polling than necessary. This aspect surfaced as a high overall MPI_poll failure rate. Straightforward reduction of the number of polls can already improve performance substantially, but statically determining the optimal polling rate is quite hard. Typically it depends on many factors (besides the host and network hardware, the messaging middleware, the application, the problem instance, etc.), but it can also change over the application's runtime. The optimized version of *process_messages* thus *dynamically* changes the polling rate, based on the actually experienced message arrival rate. Note that as the data transfers occur in an essentially unpredictable order, blocking receives at the MPI layer cannot be applied effectively, since this would introduce additional delays.

**Prioritize I/O tasks** (PRIO) – DIVINE implements timeout-based flushing of pending work to improve performance by providing other nodes with additional work in cases where message combining would otherwise postpone

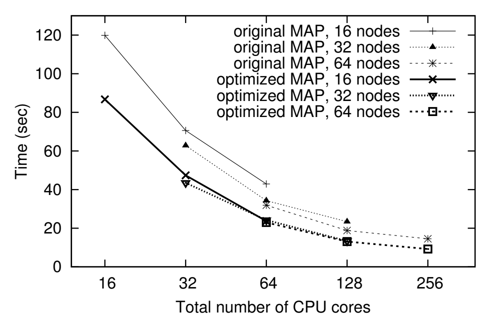


**Figure 3. Optimization effects for MAP**

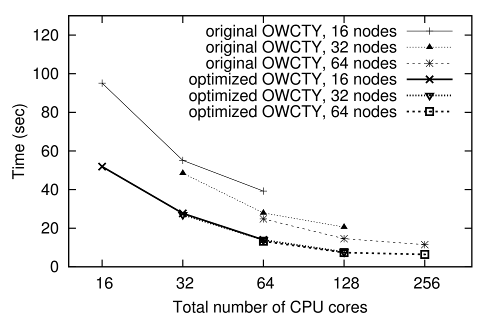


**Figure 4. Optimizations effects for OWCTY**

transfers too long. Furthermore, distributed termination detection is a requirement for DIVINE's correct functioning, but involves a separate MPI communication channel and its associated polling overhead. Both these tasks in *process_messages* were not timing-critical, yet originally were performed each time the primitive was invoked. In the optimized version, both overheads were reduced by only executing the associated code a suitably small fraction of the time.

**Optimize message flushing** (FLUSH) – Another aspect that was optimized, is the flushing of messages during the applications runtime, including distributed termination phases (as discussed in Section 2, each DIVINE algorithm has several of these phases). When running out of local work, the original implementation simply flushed *all* outstanding (non-full) messages in a fixed sequential order, which caused much congestion due to hotspots in the network and at the receivers. Also, as every message was flushed indiscriminately, the average message size during distributed termination detection phases dropped substantially, causing a relatively high overhead. Message flushing was optimized quite similarly to the Awari application [27]. In the new version, messages are now flushed from large to small – effectively spreading the traffic over the network – also taking care not to exceed a reasonable upperbound in the outgoing traffic rate – thus avoiding the syndrome of frantically sending tiny messages.

**Pre-establish network connections** (PRESYNC) – A final optimization is of a quite different nature. As will be discussed in Section 4.3, DIVINE is also suitable for running on wide-area distributed systems, but it was noticed that during large-scale grid experiments some endpoints would often fail to start communicating efficiently. Sometimes this situation could prolong such that this led to a huge backlog of MPI messages at the sender, eventually causing the application to fail due to excessive paging. The cause of the problem is that the MPI implementation used (Open MPI [14]) establishes TCP connections on-demand. This is often a useful feature, as it decreases large-scale MPI initialization time, and often only a small subset of the endpoints communicate point-to-point. However, DIVINE is a-typical in the sense that it requires *every* endpoint to communicate with every other endpoint. Also, immediately after startup, it starts communicating at peak data rates. These data rates can be such that they can fill almost the entire capacity of the wide-area network between sites, making further connection-establishment very difficult due to timeouts. This issue was resolved by the addition of a (by purpose) naively implemented small all-to-all data exchange at the initialization of DIVINE's runtime system, when the network is still uncongested. This forces all network connections to be readily available when the actual data transfers start.

## 3.2 Impact of the Optimizations

Our performance evaluation was done on the Distributed ASCI Supercomputer [7] (DAS-3), a wide-area distributed system for Computer Science research in the Netherlands. DAS-3 consists of five clusters distributed over four sites. DAS-3 uses Myri-10G networking technology from Myricom both as an internal high-speed interconnect as well as an interface to remote DAS-3 clusters. DAS-3 is largely homogeneous: every cluster uses dual-CPU AMD Opteron nodes, but with different clock speeds and/or number of CPU cores. For the single-cluster performance evaluations in this paper we used the DAS-3 cluster at VU University, since it has the largest number of compute nodes and cores (85 nodes with a dual-cpu, dual-core 2.4 GHz AMD Opterons).

Figures 3 and 4 show the optimization effects for MAP and OWCTY on an increasing number of DAS-3/VU compute nodes, using Myri-10G's native MX layer for communication. Results are shown for 1, 2 and 4 cores per compute node. We used LTL problem instance 6 of the Anderson specification from the BEEM benchmark set [23] (this

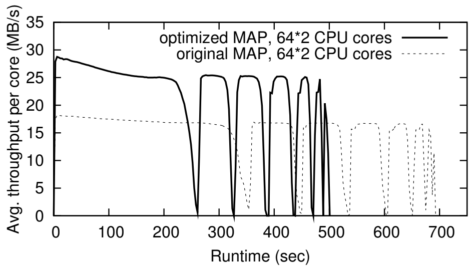


**Figure 5. Per-core throughput of MAP**

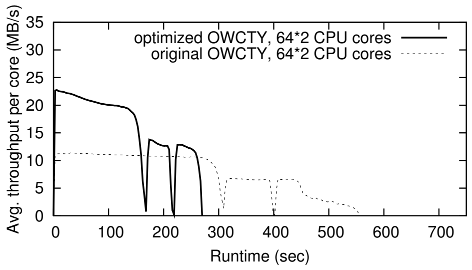


**Figure 6. Per-core throughput of OWCTY**

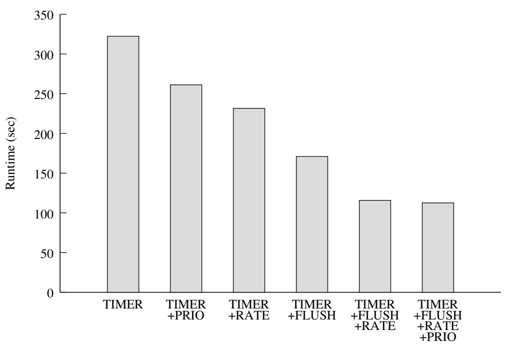


**Figure 7. Impact of the individual optimizations for OWCTY on 64*4 cores**

| Nodes | Total cores | Runtime (s) MAP | Runtime (s) OWCTY | Efficiency MAP | Efficiency OWCTY |
|---|---|---|---|---|---|
| 1 | 1 | 956.8 | 628.8 | 100% | 100% |
| 16 | 16 | 73.9 | 42.5 | 81% | 92% |
| 16 | 32 | 39.4 | 22.5 | 76% | 87% |
| 16 | 64 | 20.6 | 11.4 | 73% | 86% |
| 64 | 64 | 19.5 | 10.9 | 77% | 90% |
| 64 | 128 | 10.8 | 6.0 | 69% | 82% |
| 64 | 256 | 7.4 | 4.3 | 51% | 57% |

**Table 1. Efficiency of MAP and OWCTY**

specification concerns the correctness of a mutual exclusion algorithm). In this case the state space has to be searched completely, but it is still small enough to fit into the memory of 16 compute nodes, making a scalability comparison feasible. There are several conclusions to be drawn from these figures:

- The performance for both the MAP and the OWCTY implementation has increased substantially: they are about 30–50% faster than before;
- The scalability of both algorithms is quite good: by increasing the number of cores with a factor 16 (from 16 to 256 cores), both MAP and OWCTY run about a factor 10 faster;
- Due to reduced overheads, the performance of the optimized version is almost insensitive to the placement of multiple processes on the same node, unlike the original version.

Figures 5 and 6 illustrate the effect of the optimizations on the applications' throughput as a function of their runtime. In this case a larger instance of the Anderson problem was used to obtain a higher resolution graph.

In the case of MAP (Figure 5), the throughput graph clearly shows the application-specific phases. The first phase (originally taking 354 seconds, optimized 261 seconds), the state space is constructed on-the-fly, besides applying the MAP algorithm itself (see Section 2.2). It is therefore taking significantly longer than the subsequent phases. The graph clearly shows that peak throughput is maintained almost throughout the application's runtime, and that the optimizations have let the average per-core throughput increase from 15.0 to 22.1 MByte/s.

Likewise, the throughput graph for OWCTY (Figure 6) shows the application-specific phases. As in the case for MAP, in the first phase (lasting resp. 309 and 168 seconds) the state space is constructed on-the-fly. After that, OWCTY here only requires two smaller phases to complete (as is true for many specifications, see Section 2.2). However, note the long tail of the last phase, which is due to inefficiencies in the original termination detection implementation. For OWCTY, the average per-core throughput increases from 7.9 to 16.4 MByte/s.

To investigate the *relative* impact of the optimizations, we constructed a version of DiVinE where combinations of individual optimizations discussed above could be enabled dynamically at runtime. The results for OWCTY using 256 cores on the larger instance of the Anderson model are shown in Figure 7. The version with TIMER optimization was used as a baseline, since this modification is re-

**Table 2. Large-scale models used**

| Model | Description | Specification | State space |
|---|---|---|---|
| Elevator | Elevator controller correctness | elevator.4.prop2 (scaled) | 123.8 GB |
| Publish-subscribe | Groupware protocol | public-subscribe.5.prop1 | 209.7 GB |
| AT | Timing based mutual exclusion | at.7.prop2 | 245.0 GB |
| Le Lann | Token ring leader election | lann.8.prop1 | > 320 GB |
| GIOP | CORBA General Inter-Orb Protocol | scenario 1, property 3 | 203.8 GB |
| Lunar | Ad hoc routing protocol | scenario 4d; two properties | 181.6 GB |

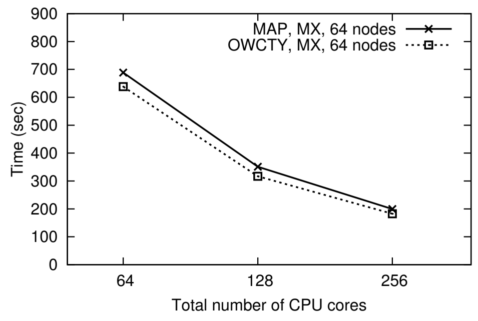


**Figure 8. Publish-subscribe on 64 nodes**

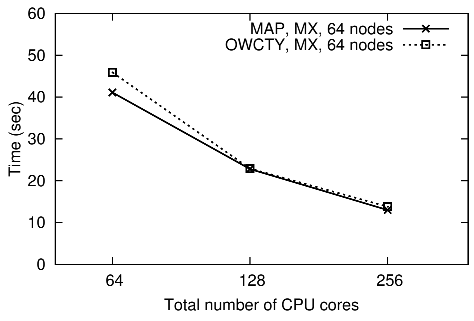


**Figure 9. Lunar on 64 nodes, property 1**

quired to still get reasonable speedup as the number of CPU cores grows. As shown, optimization FLUSH has the highest impact on performance, but RATE and PRIO also have significant impact. Optimization PRESYNC (not shown) only has impact when the network contains a bottleneck, such as for grid configurations discussed in Section 4.3.

It should be noted that some of the optimizations are not independent: optimization RATE can by itself reduce the polling rate such that optimization PRIO becomes less effective (or required). Interestingly, both polling optimizations turn out to have less impact for MAP, but this is explained by the fact that MAP already applies an (ad-hoc) polling rate reduction at the application level. However, for both OWCTY and MAP, when a network with higher host overhead is employed (e.g., a TCP/IP network as discussed later in this paper), enabling both RATE and PRIO is still required to obtain good performance.

To estimate the efficiency of MAP and OWCTY, we ran single-core versions on a special DAS-3/VU node equipped with 16 GB memory (the regular compute nodes have 4 GB, which is insufficient to store the state space there). The results are shown in Table 1. Considering the high data exchange rates of the fine-grained applications, and the highly demanding traffic pattern (irregular all-to-all), these results can be considered quite good. Also note that this is still a reasonably small problem: DiVinE's efficiency increases further with problem size as the relative impact of synchronizations between the application phases then lessens.

## 4 Scalability of Optimized DiVinE

Besides being able to analyze medium-scale models efficiently, another important use case for DiVinE is dealing with problems that simply are too large to fit into the main memory of (typical) small-scale computer systems. In this section we will therefore focus on the performance of MAP and OWCTY on a diverse set of large-scale models. We will look into scalability aspects, and also examine the impact of network overhead.

An overview of the model specifications used is shown in Table 2. The first four models (Elevator, Publish-subscribe, AT, and Le Lann) are written in DiVinE's native modeling language "DVE", and are taken from the on-line BEEM database [23]. The last two models are examples of realistic specifications written in Promela, the SPIN modeling language. These models are related to protocols for the CORBA architecture (General Inter-Orb Protocol, GIOP [17]) and Ad hoc routing (Lunar [29]), for which we examine two different LTL properties. In DiVinE, Promela specifications are handled using the embedded "NIPS" module. NIPS is a complete reimplementation of the original SPIN tool, by means of a specially developed model-checking *virtual machine* [28].

In two models, AT and Le Lann, the LTL formula being verified is false, i.e., there exists a counterexample that has to be found by the DiVinE tool. In all other cases, the LTL formula provided is valid in the model, as a result of

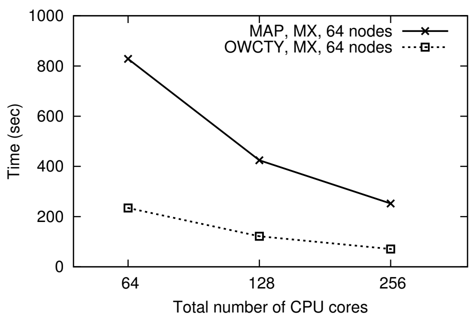


**Figure 10. Elevator on 64 nodes**

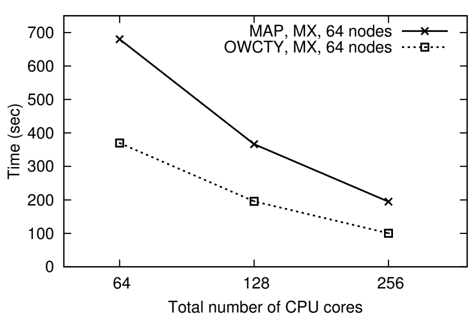


**Figure 11. GIOP on 64 nodes**

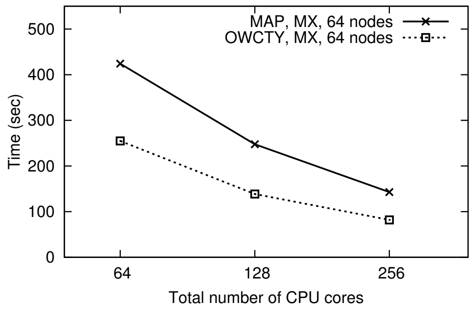


**Figure 12. Lunar on 64 nodes, property 2**

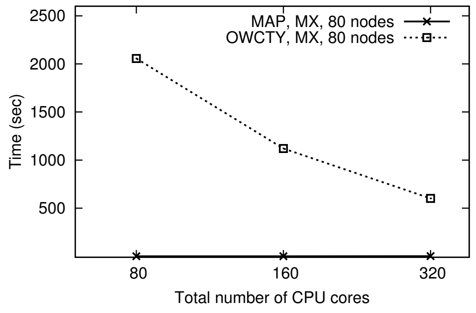


**Figure 13. AT on 80 nodes**

which the entire state space has to be built and analyzed for the presence of accepting cycles. The state space memory requirements shown are the ones reported by OWCTY; the algorithm-dependent per-state memory overhead for MAP is somewhat lower, reducing its overall memory requirements on average by 14%.

## 4.1 DAS-3 Cluster Performance

We will now show results of 1-, 2- and 4-core configurations of the DAS-3/VU cluster, using MX over Myri-10G like in the previous section. We used 64 nodes unless (when noted) the search space was so big that 80 nodes were required to store it in main memory.

The specifications requiring a full space search are shown in Figures 8 – 12. Interestingly, two groups of specifications with similar performance patterns can be identified: for Publish-subscribe and the first Ad hoc routing specification, OWCTY and MAP are about equally fast. In contrast, for Elevator, GIOP and the second Ad hoc routing specification, OWCTY is much faster than MAP, but both show good scalability when increasing the number of cores. The reason is that in the case of Publish-subscribe, MAP only requires 9 very short cycle-searching phases after the first on-the-fly one. For the second Ad hoc routing specification, the property is even known without either OWCTY or MAP having to start any additional phases, therefore their performance is about equal. This should be contrasted with the second group of specifications, where, e.g., MAP on Elevator requires 21 longer phases and OWCTY only needs three phases.

The two inconsistent specifications show a rather different picture. As seen in Figure 13, MAP is extremely quick in finding the counterexample: it is at the bottom of the graph, taking only a few seconds. OWCTY requires a very expensive preparation phase constructing the entire search space, which is large enough that 80 compute nodes are required. The Le Lann specification (graph not included) even has a search space too large to fit on DAS-3/VU so OWCTY is unable to find the counterexample. On the other hand, like for AT, MAP is able to find it in a matter of seconds.

## 4.2 Network Impact

In this section we compare DiVinE's performance using Myri-10G's native MX interface with performance using TCP/IP over the same network. We use TCP/IP to provide insight into DiVinE's performance on a higher-overhead

| MPI primitive | MX<br>64 cores | MX<br>128 cores | MX<br>256 cores | TCP<br>64 cores | TCP<br>128 cores | TCP<br>256 cores |
|---|---|---|---|---|---|---|
| MPI_Isend | 12.5 | 13.1 | 13.4 | 18.6 | 19.4 | 19.8 |
| MPI_Recv | 7.9 | 8.3 | 8.8 | 7.7 | 7.6 | 7.3 |
| MPI_Iprobe (failed) | 1.9 | 2.6 | 4.6 | 38.7 | 51.2 | 87.7 |
| MPI_Iprobe (success) | 4.2 | 4.5 | 5.1 | 3.2 | 3.7 | 4.9 |

**Table 3. Average MPI host overhead in $\mu$s on DAS-3/VU**

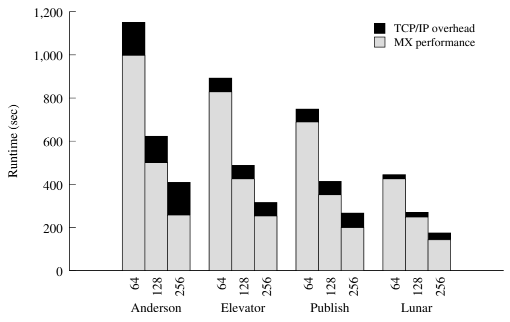


**Figure 14. TCP/IP overhead for MAP**

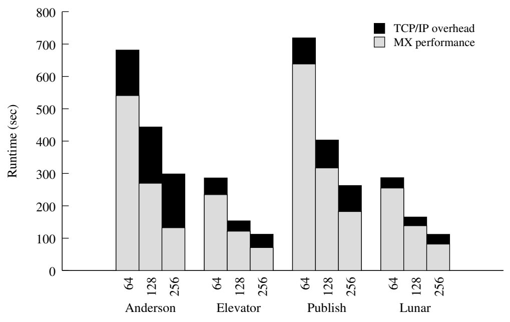


**Figure 15. TCP/IP overhead for OWCTY**

network, much as would be the case on a general purpose 1Gb/s or 10Gb/s Ethernet. We use the same version of Open MPI for both MX and TCP/IP, as Open MPI conveniently allows selection of a specific network backend at runtime.

Table 3 shows host-level overheads using MX and TCP/IP for the most important MPI primitives used in DiVinE. The overheads were measured using the MAP application (results using OWCTY are very similar). Receives and successful probes have about the same overhead, but send overhead is about 50% higher on TCP/IP. The biggest difference is for failing probes, however, where TCP/IP is about a factor 20 more expensive than MX. In addition, kernel-based TCP/IP receive processing is a source of overhead, but this is harder to measure as it occurs interrupt-driven, asynchronous to the application. Note that differences in end-to-end latency and peak throughput are less relevant, as they have little impact on application performance given DiVinE's asynchronous communication style.

Figures 14 and 15 show a quite consistent pattern in the impact of using TCP/IP instead of MX. The TCP/IP interface with higher send and receive overhead does increase the runtimes, but interestingly enough this increase is almost independent of the number of cores used. It should be noted that in DiVinE the communication rate *per core* is in principle independent of the total number of cores, so one would expect that doubling the number of cores would on average halve the total number of sends and receives per core. Unfortunately, the overhead of a TCP/IP-based network is significantly more dependent on the number of remote endpoints than MX, as shown above, which counterbalances the gain due to the reduced total number of transfers per core.

## 4.3 DAS-3 Grid Performance

Given DiVinE's consistent use of asynchronous communication throughout its execution, an interesting question is to what extent its overall performance is truly latency independent. However, running the distributed model checker at a large scale does pose very high demands on the wide-area network bandwidth, as every compute node indiscriminately needs to transfer a large portion of its protocol messages to nodes at other clusters. Fortunately, DAS-3 provides the opportunity to examine this aspect in detail since it features a dedicated wide-area network called StarPlane [24], built out of multiple optical 10G links. The impact of using a single or multiple optical links on distributed application performance (including DiVinE) is discussed in another recent paper [20]; here we will use a static configuration of two 10G links between the sites.

We use a DAS-3 grid configuration of 160 compute nodes distributed over 4 clusters, located at 3 sites in the Netherlands (VU University, University of Amsterdam and Leiden University). The one-way latencies over TCP/IP between these clusters range between 0.37 and 0.98 milliseconds, which should be contrasted with an intra-cluster one-way TCP/IP latency on DAS-3/VU of 26 microseconds, i.e., up to a factor 38 difference.

Figures 16 and 17 show the results for running an increasingly large instance of the Elevator specification on

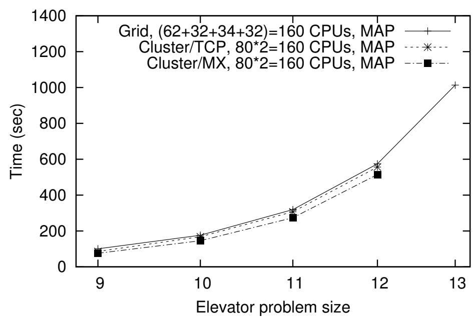


**Figure 16. Elevator/MAP on a grid**

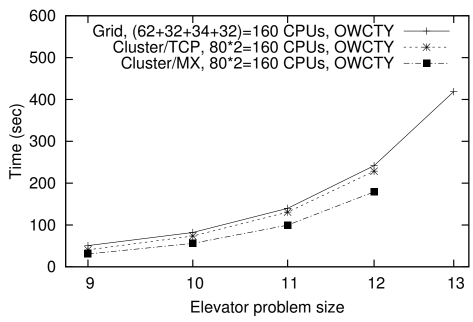


**Figure 17. Elevator/OWCTY on a grid**

both the grid and a DAS-3/VU cluster. Note that problem size 13 is too big to be run on the single DAS-3/VU cluster alone. The figures show that despite the additional wide-area latencies incurred, for both MAP and OWCTY grid performance is actually very close to single-cluster performance using the same TCP/IP protocol stack.

## 5 Discussion and Conclusions

The work on distributed-memory verification is quite extensive, and growing in recent years. In this paper we discussed a distributed-memory tool for enumerative model checking. Distributed-memory techniques have also been applied in other verification areas, e.g., verification of timed systems [3], equivalence checking [4], state space construction [13], and $\mu$-calculus model checking [5]. However, these systems have thus far not been evaluated on and optimized for large-scale clusters (or grids). A possible exception is the *symbolic* model checker in [12], but no scalability results are reported.

Our paper is novel in bringing distributed model checking closer to industrial applications. Both the scale and efficiency with which we are now able to verify very large systems is to the best of our knowledge without precedent. For very large-scale models, state space reduction may still remain necessary, but this can often be orthogonal to the techniques discussed in this paper. For example, for partial order reduction, a *static* transformation approach is known [18], but distributed algorithms also exist [6]. State compression is another popular technique that directly applies in a distributed context. Keeping parts of the state space on secondary storage, while still maintaining good performance, is also a possibility [15].

We have discussed two main distributed algorithms in DIVINE, and we have shown how several optimizations together improved their performance by 30 to 50%. We compared the performance of these two algorithms on seven representative large models, having quite different characteristics. We have shown that the optimizations allow the algorithms to scale well, up to at least 256 cores, efficiently exploiting current multi-core architectures. However, as *many-core* is an inevitable trend in computer architecture, it appears likely that at some point a single-address-space multithreaded implementation should be integrated with the current version for best performance [2].

Some of the optimizations discussed are not unique to DIVINE, but will also be applicable in other distributed applications. For example, the auto-tuning polling rate optimization described will be useful in several cases where applications have to employ non-blocking polling due to the irregularity of the communication patterns [27].

The performance differences shown can be used to plan an efficient model checking workflow, during the development of an abstract specification. If a property of a model is expected to hold, and the state space fits completely into (distributed) memory, the OWCTY algorithm will typically be preferable as it can give up to three times faster results than MAP. However, if the status of a property is uncertain, MAP will generally be preferable instead, as it works on-the-fly, and may thus find counterexamples quickly (even when the entire state space would not fit into memory). Also, if a property holds after all, MAP will still perform quite well due to its good scalability.

In this paper, we also analyzed the sensitivity of the model checking algorithms to network protocol overhead, and we have shown how the consistent use of asynchronous communication even allows efficiently running the model checker on a large-scale computational grid. This thus enables further scaling up the model checker for realistic use cases, where the state space to be examined quickly grows even beyond the capacity of a single large compute cluster.

**Note –** DIVINE is available from *http://divine.fi.muni.cz*. The cluster-based tools, containing the optimizations discussed, are now part of DIVINE-CLUSTER, distinguishing them from other instances of DIVINE.

## References

[1] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *FMCO'05*, volume 4590 of *LNCS*, pages 281–293. Springer, 2006.

[2] J. Barnat, L. Brim, and P. Rockai. Scalable Multi-core LTL Model-Checking. In *SPIN'07*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[3] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed Timed Model Checking — How the Search Order Matters. In *CAV'00*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

[4] S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *STTT*, 7(1):74–86, 2005.

[5] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free mu-Calculus. In *TACAS'01*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[6] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed Partial Order Reduction. *Electronic Notes in Theoretical Computer Science*, 128:63–74, 2005.

[7] F. Cappello and H.E. Bal. Toward an International "Computer Science Grid" (keynote). In *CCGrid'07*, pages 3–12, 2007.

[8] I. Černá and R. Pelánek. Relating Hierarchy of Temporal Properties to Model Checking. In *MFCS'03*, volume 2747 of *LNCS*, pages 318–327. Springer, 2003.

[9] C. Courcoubetics, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *CAV'91*, pages 233–242. Springer, 1991.

[10] E.W. Dijkstra. Shmuel Safra's version of termination detection. EWD Manuscripts, no. 998, January 1987.

[11] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-detection Algorithm? In *TACAS'01*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[12] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Verifying Very Large Industrial Circuits Using 100 Processes and Beyond. *Int. J. Found. Comput. Sci.*, 18(1), 2007.

[13] H. Garavel, R. Mateescu, and I.M. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN'01*, volume 2057 of *LNCS*, pages 216–234. Springer, 2001.

[14] R.L. Graham, T.S. Woodall, and J.M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proc. 6th Int. Conf. on Par. Proc. and Appl. Math.*, pages 228–239, Poznan, Poland, September 2005.

[15] M. Hammer and M. Weber. "To Store or Not To Store" Reloaded: Reclaiming Memory on Demand. In *Formal Methods: Application and Technology (FMICS'2006)*, volume 4346 of *LNCS*, pages 51–66. Springer, 2007.

[16] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[17] M. Kamel and S. Leue. Formalization and Validation of the General Inter-Orb Protocol (GIOP) using Promela and SPIN. In *In: Software Tools for Technology Transfer*, pages 394–409. Springer, 2000.

[18] R.P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Combining Software and Hardware Verification Techniques. *Form. Methods Syst. Des.*, 21(3):251–280, 2002.

[19] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, March 2007.

[20] J. Maassen, K. Verstoep, H.E. Bal, P. Grosso, and C. de Laat. Assessing the Impact of Future Reconfigurable Optical Networks on Application Performance. In *IPDPS'09: 6th High-Performance Grid Computing Workshop (HPGC 2009)*, 2009.

[21] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.

[22] R. Pelánek. Typical Structural Properties of State Spaces. In *SPIN'04*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[23] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[24] StarPlane project. http://www.starplane.org.

[25] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, January 1972.

[26] M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 322–331. Computer Society Press, 1986.

[27] K. Verstoep, J. Maassen, H.E. Bal, and J.W. Romein. Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed. In *CCGrid'08*, 2008.

[28] M. Weber. An Embeddable Virtual Machine for State Space Generation. In *SPIN'07*, pages 168–186, 2007.

[29] O. Wibling, J. Parrow, and A. Neville Pears. Automatized Verification of Ad Hoc Routing Protocols. In *FORTE'04*, volume 3235 of *LNCS*, pages 343–358. Springer, 2004.

# A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties[⋆]

Jiří Barnat, Luboš Brim, and Petr Ročkai[⋆⋆]

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

**Abstract.** One of the most important open problems of parallel LTL model-checking is to design an on-the-fly scalable parallel algorithm with linear time complexity. Such an algorithm would give the optimality we have in sequential LTL model-checking. In this paper we give a partial solution to the problem. We propose an algorithm that has the required properties for a very rich subset of LTL properties, namely those expressible by weak Büchi automata.

## 1 Introduction

Formal verification is nowadays an established part of the design methodology in many industrial applications. Moreover, it is no more regarded only as a supplementary vehicle to more traditional coverage oriented testing and simulation activities, ruther it takes in many situation the role of the primary validation technique. In [14] the authors report about replacing testing with symbolic verification in the recent Intel Core i7 processor design.

Traditional verification techniques are computationally demanding and memory-intensive in general and their scalability to extremely large and complex systems routinely seen in practice these days is limited. Verifying complex systems with a high degree of fidelity implies exceedingly large state spaces that must be analyzed. These state spaces are typically too large to fit into memory of a single contemporary computer, unless substantial simplification leading to removal of important features from the model are made. One solution to deal with the memory problems is to use more powerfull parallel computers. Enormous recent progress in hardware architectures, which has measured several orders of magnitude with respect to various physical parameters such as computing power, memory size at all hierarchy levels from caches to disks, power consumption, networking, physical size and cost, has made parallel computers easily available. On the other hand, this architectural shift requires introducing algorithmic changes

[⋆] This work has been partially supported by the Czech Science Foundation grants No. 201/09/1389 and P201/09/P497, and by the Academy of Sciences grants No. 1ET408050503 and 1ET400300504.

[⋆⋆] The third author has been partially supported by Red Hat, Inc.

to our tools. Without them we will not be able to fully utilize the power of parallel computers.

In this paper we consider parallel explicit-state LTL model-checking. Explicit-state model checking is a branch of model checking in which the states and transitions are stored explicitly as the model checking program traverses through the state space. The main practical problem with explicit model checking is the state space explosion. To reduce the state explosion effect, explicit model checking works on-the-fly to gradually generate and check the state space, being thus able to find a counter-example without ever constructing the complete state space.

In the case of automata-based approach to explicit-state LTL model-checking the verification problem is reduced to checking the non-emptiness of a Büchi automaton, hence the detection of a reachable accepting cycle in a rooted directed graph. The best known on-the-fly algorithms use depth-first-search (DFS) strategies.

It is well-known that DFS based algorithms are difficult to parallelize. For this reason parallel explicit-state LTL model-checking algorithms rely on other state exploration strategies than DFS. Typically, they use some variant of breath-first-search (BFS) strategy, which is well suited for parallelization. Several different algorithms have been proposed for parallel explicit-state LTL model-checking. Contrary to the serial case, it is difficult to identify the best algorithm among them. One of the reasons is that some of these algorithms have higher time complexity, but work on-the-fly, while others are on-the-fly with worse time complexity.

One of the main open problems in explicit-state LTL model-checking is to develop a parallel algorithm that works on-the-fly and has linear time complexity. In this paper we propose a parallel on-the-fly linear algorithm for LTL model-checking of weak LTL properties. Weak LTL properties are those that are expressible by weak Büchi automata, i.e. automata in which there is no cycle with both accepting and non-accepting state on its path. The studies of temporal properties [8,5] reveal that verification of up to 90% of LTL properties leads to a weak case. The most common weak LTL properties are the response properties, e.g. properties stating that whenever A happens, B happens eventually. An important aspect of our approach is that there is no difference in handling weak and non-weak LTL formulas. However, if it is required, we can perform test for a weak case within the model checking procedure with no impact on both theoretical complexity and practical performance.

Our algorithm extends the linear parallel OWCTY algorithm [5] by a heuristic for early accepting cycle discovery. The heuristic is based on the MAP algorithm [4], in partiucalr it employs the fact that if an accepting state is its own predecessor, it lies on an accepting cycle. The new algorithm thus combines the basic OWCTY algorithm with a limited propagation of selected accepting states as performed within MAP algorithm.

The new algorithm is able to detect accepting cycle and produce the so called counter-example without constructing the entire state space, hence it can be classified as on-the-fly algorithm. Since it relies on a heuristic method, a natural question is how much on-the-fly the algorithm actually is. Unfortunately, there is no standard way to compare LTL model-checking algorithms regarding this aspect. For DFS-based sequential algorithms the question is easier to answer and has been discussed by several authors. For parallel algorithms the situation is much complicated. Therefore, we identify some simple criteria for the degree of "on-the-flyness" of an algorithm, and subsequently classify our algorithm according these criteria.

Our new algorithm has been implemented in the multi-core version of the parallel LTL model-checker DiVinE [3,2]. The tool is available from its webpage [7] and is also distributed as a part of Fedora 11 release.

We proceed as follows: Section 2 establishes the necessary notions used in the algorithm. Section 3 then presents the algorithm itself. Section 4 discusses the on-the-fly notion in more detail and also contains discussion on related work. Section 5 reports results on experimental evaluation of the algorithm, and Section 6 contains the conclusions and an open questions.

## 2 Preliminaries

Automata-theoretic approach to explicit-state LTL model-checking [19] exploits the fact that every set of executions expressible by an LTL formula can be described by a *Büchi automaton*. In particular, the approach suggests to express all system executions by a *system automaton* and all executions not satisfying the formula by a *property* or *negative claim automaton*. These automata are combined into their synchronous product in order to check for the presence of system executions that violate the property expressed by the formula. The language recognized by the *product automaton* is empty if and only if no system execution is invalid.

The language emptiness problem for Büchi automata can be expressed as an *accepting cycle detection problem* in a graph. Each Büchi automaton can be naturally identified with an *automaton graph* which is a directed graph $G = (V, E, s, A)$ where $V$ is the set of states ($n = |V|$), $E$ is a set of edges ($m = |E|$), $s$ is an initial state, and $A \subseteq V$ is a set of accepting states. We say that a cycle in $G$ is accepting if it contains an accepting state. Let $\mathcal{A}$ be a Büchi automaton and $G_{\mathcal{A}}$ the corresponding automaton graph. Then $\mathcal{A}$ recognizes a nonempty language iff $G_{\mathcal{A}}$ contains an accepting cycle reachable from $s$. The LTL model-checking problem is thus reduced to the accepting cycle detection problem in the automaton graph.

The optimal sequential algorithms for accepting cycle detection use depth-first search strategies to detect accepting cycles. The individual algorithms differ in their space requirements, length of the counter-example produced, and other aspects. For a recent survey we refer to [18]. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best suitable algorithm for explicit-state sequential LTL model checking. The algorithm was proposed by Courcoubetis et al. [6] and its main idea is to use two interleaved searches to detect reachable accepting cycles. The first search discovers accepting states while the second one, the nested one, checks for self-reachability. Several modifications of the algorithm have been suggested to remedy some of its disadvantages [12]. Another group of optimal algorithms are *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into strongly connected components (SCCs) [17]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counter-examples in general. For a survey we refer to [16]. The time complexity of all these algorithms is linear in the size of the graph, i.e. $\mathcal{O}(m + n)$, where $m$ is the number of edges and $n$ is the number of states.

The effectiveness of the *Nested DFS* algorithm is achieved due to the particular order in which the graph is explored and which guarantees that states are not re-visited more than twice. In fact, all the best-known algorithms rely on the same exploring principle, namely the *postorder* as computed by the DFS. It is a well-known fact that the postorder problem is P-complete and, consequently a scalable parallel algorithm which would be directly based on DFS postorder is unlikely to exist.

Several solutions to overcome the postorder problem in a parallel environment have been suggested. The parallel algorithms were developed employing additional data structures and/or different search and distribution strategies. In the next section we present two of them. For a survey on other algorithms we refer to [1].

## 3 Algorithm

The proposed algorithm combines the OWCTY [5] approach with a heuristic for early accepting cycle discovery based on the MAP algorithm [4].

The basic OWCTY algorithm uses topological sort for cycle detection – a linear time algorithm that does not depend on DFS postorder and can thus be parallelized reasonably well. However, topological sort algorithm cannot detect *accepting cycles* as such. Therefore, the OWCTY algorithm uses other provisions to eliminate detection of non-accepting cycles. In particular, the algorithm computes a set of states predecessed by an accepting cycle, the so called approximation set. If the algorithm terminates and the set is empty, there is no accepting cycle in the graph. The set is computed in several phases as follows. First, a phase called Initialize is executed to explore the complete state space of the automaton and to set up internal data for use by subsequent phases. Note that all reachable states are initially part of the approximation set. This phase is the one where we apply our "on-the-fly" heuristics. The latter two phases are called Elim-No-Accepting and Elim-No-Predecessors. These phases remove states from the approximation set that cannot be part of an accepting cycle. They are executed repeatedly until a fix-point is reached. An important observation is that if the underlying automaton graph is weak (system automaton was producted with weak negative claim Büchi automaton), the phases need to be executed exactly once. Further details of the algorithm and its phases can

**Algorithm 1.** DetectAcceptingCycle

```
Require: Implicit definition of G=(V,E,ACC)
1: Initialize()
2: $oldSize \leftarrow \infty$
3: while $(ApproxSet.size \neq oldSize) \wedge (ApproxSet.size > 0)$ do
4:     $oldSize \leftarrow ApproxSet.size$
5:     Elim-No-Accepting()
6:     Elim-No-Predecessors()
7: return $ApproxSet.size > 0$
```

**Algorithm 2.** Initialize

```
1: $s \leftarrow$ GetInitialState()
2: $ApproxSet \leftarrow \{s\}$
3: $ApproxSet.setMap(s, 0)$
4: $Open.pushBack(s)$
5: while $Open.isNotEmpty()$ do
6:     $s \leftarrow Open.popFront()$
7:     for all $t \in$ GetSuccessors$(s)$ do
8:         if $t \notin ApproxSet$ then
9:             $ApproxSet \leftarrow ApproxSet \cup \{t\}$
10:            $Open.pushBack(t)$
11:        if IsAccepting$(t)$ then
12:            if $t = s \vee ApproxSet.getMap(s) = t$ then
13:                AcceptingCycleFound()
14:                return true
15:            $ApproxSet.setMap(t, \textsc{max}(t, ApproxSet.getMap(s)))$
16:        else
17:            $ApproxSet.setMap(t, ApproxSet.getMap(s))$
```

be found in [5]. For clarity, we just list the pseudo-code of the new combined algorithm.

The original MAP algorithm is based on propagation of maximum accepting predecessors and, similarly to OWCTY, its execution is organized into multiple passes. Each pass fully propagates (this includes re-propagation) maximum (according to given order) accepting predecessors of all states. Even a single pass of such algorithm is super-linear, up to $n$ passes may need to be executed. After each pass, states constituting maximum accepting predecessors are marked as non-accepting and next pass is executed. The MAP algorithm finishes when a state is found to be its own maximum accepting predecessor (this means that an accepting cycle has been discovered in the state space), or when there are no reachable accepting states.

The idea of propagating one accepting predecessor along all newly discovered edges is at heart of the proposed heuristic extension of OWCTY. If the propagated accepting state is propagated into itself, an accepting cycle is discovered

**Algorithm 3.** ELIM-NO-ACCEPTING

```
 1: $ApproxSet' \leftarrow \emptyset$
 2: for all $s \in ApproxSet$ do
 3:     if IsAccepting($s$) then
 4:         $Open.pushBack(s)$
 5:         $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
 6:         $ApproxSet'.setPredecessorCount(s, 0)$
 7: $ApproxSet \leftarrow ApproxSet'$
 8: while $Open.isNotEmpty()$ do
 9:     $s \leftarrow Open.popFront()$
10:     for all $t \in$ GetSuccessors($s$) do
11:         if $t \in ApproxSet$ then
12:             $ApproxSet.increasePredecessorCount(t)$
13:         else
14:             $Open.pushBack(t)$
15:             $ApproxSet \leftarrow ApproxSet \cup \{t\}$
16:             $ApproxSet.setPredecessorCount(t, 0)$
```

**Algorithm 4.** ELIM-NO-PREDECESSORS

```
 1: for all $s \in ApproxSet$ do
 2:     if $ApproxSet.getPredecessorCount(s) = 0$ then
 3:         $Open.pushBack(s)$
 4: while $Open.isNotEmpty()$ do
 5:     $s \leftarrow Open.popFront()$
 6:     $ApproxSet \leftarrow ApproxSet \setminus \{s\}$
 7:     for all $t \in$ GetSuccessors($s$) do
 8:         $ApproxSet.decreasePredecessorCount(t)$
 9:         if $ApproxSet.getPredecessorCount(t) = 0$ then
10:             $Open.pushBack(t)$
```

and the computation is terminated. Likewise the MAP algorithm, an accepting state to be propagated is selected as a maximal accepting state among all accepting states visited by the traversal algorithm on a path from the initial state of the graph to the currently expanded state. Since the INITIALIZE phase of OWCTY needs to explore full state space, we can employ it to perform limited accepting cycle detection using maximal accepting state propagation. Unlike the MAP algorithm, we however avoid any re-propagation to keep the INITIALIZE phase complexity linear in the size of the graph. This means that some accepting cycles that would be actually discovered using re-propagation, may be missed. In particular, there are three general reasons for not discovering an accepting cycle with our heuristics. First, the maximum accepting predecessor of the cycle may not lie on the cycle itself, see Figure 1(a). Second, the maximum accepting predecessor value does not reach the originating state due to the absence of a fresh path (path made of yet unvisited states), see Figure 1(b). And third, the

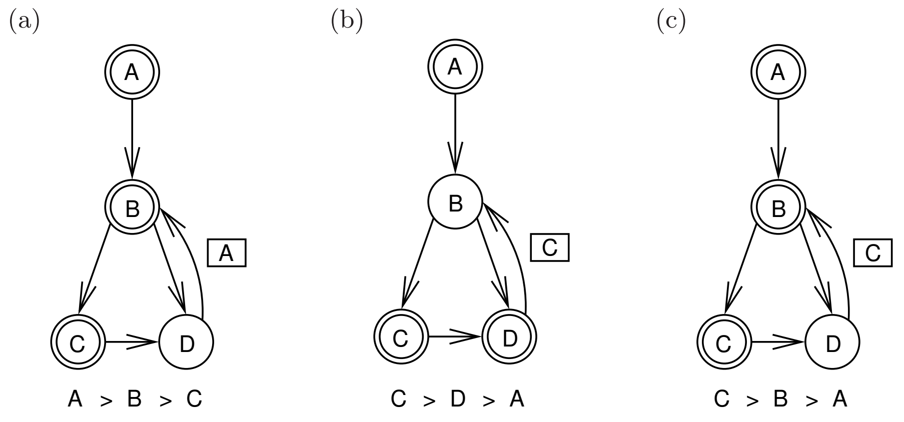


**Fig. 1.** Three scenarios where no accepting cycle will be discovered using accepting state propagation. a) Maximal accepting predecessor is out of the cycle. b) There is no fresh path back to the maximal accepting state. c) Wrong order of propagation, $C \to D$ is explored before $B \to D$, hence, $C$ is propagated from $D$.

maximum accepting predecessor value does not reach the originating state due to a wrong propagation order, see Figure 1(c).

When the algorithm encounters an accepting state that is being propagated, it terminates early, producing a counter-example. On the other hand, if the Initialize phase of OWCTY fails to notice an accepting cycle, the rest of the original OWCTY algorithm is executed. Either the algorithm finds an accepting cycle (and again, produce a counter-example) or, it proves that there are no accepting cycles in the underlying graph.

An interesting feature of our algorithm is a possibilty to propagate more values simultaneously. Generally, the more values are propagated the more successful the Initialize phase might be in discovering accepting cycles. Consider for example the case $(a)$ in Figure 1. If two largest accepting states are propagated, $A$ and $B$ in this case, the cycle would be detected. Similarly, if the algorithm considers multiple distinct orderings and propagates maximal accepting states for each of them, the cycle in the case $(a)$ in Figure 1 could be detected. This would, however, require $B$ to be a maximal accepting state for some ordering.

## 4 On-the-Fly Verification

In automated verification, parallel techniques both for symbolic and explicit state approaches have been considered. While the symbolic set representations, which often employs canonical normal forms for propositional logic like BDDs, have been a breakthrough in the last decade (with the capacity to handle spaces of the size $10^{20}$ and beyond), they often turned out to not scale well with the problem sizes. Moreover, the success of their application to a given verification problem cannot be estimated in advance, since neither the size of the system in terms of lines of code nor other known metrics for the system size have proved to be useful for such estimates. Moreover, the use of BDDs is often sensible to the used variable ordering, which is sometimes difficult to determine.

For this reason, SAT-based model checking, in particular in the forms of bounded model checking and equivalence checking have recently become very popular. They still benefit from the use of symbolic methods, but tend to be more scalable as they no longer rely on canonical normal forms like BDDs. Many algorithms used in SAT solvers could also benefit from parallel processing capabilities, even though this has not yet been a topic of the mainstream research.

An alternative is the use of explicit state set representations. Clearly, for most real world systems, the state spaces are far too big for a simple explicit representation. However, many techniques like partial order reduction approach have been developed to reduce the state spaces to be examined. In contrast to symbolically represented state sets, explicit state space representations can directly benefit from multiprocessor systems and explicit state based model checking scales very well with the number of available processors.

Let alone partial order reduction techniques, another important method for coping with the state explosion problem in explicit state model checking, is the so called *on-the-fly* verification. The idea of the on-the-fly verification builds upon an observation that in many cases, especially when a system does not satisfy its specification, only a subset of the system states need to be analyzed in order to determine whether the system satisfies a given property or not. On-the-fly approaches to model checking (also reffered to as local algorithmic approaches) attempt to take advantage of this observation and construct new parts of the state space only if these parts are needed to answer the model checking question.

As mentioned in Section 2 explicit-state automata-theoretic LTL model checking relies on three procedures: the construction of an automaton that represents the negation of the LTL property (negative-claim automaton), the construction of the state space, i.e. the product automaton of system and negative-claim automata, and the check for the non-emptiness of the language recognized by the product automaton.

An interesting observation is that only those behaviors of the examined system are present in the product automaton graph that are possible in the negative-claim automaton. In other words, by constructing the product automaton graph the system behaviors that are not relevant to the validity of the verified LTL formula are pruned out. As a result, any LTL model checking algorithm that builds upon exploration of the product automaton graph may be considered as an on-the-fly algorithm. We will denote such an algorithm as Level 0 on-the-fly algorithm in the classification given below.

When the product automaton graph is constructed, an accepting cycle detection algorithm is employed for detection of accepting cycles in the product automaton graph. However, it is not necessary for the algorithm to have the product automaton constructed before it is executed. On the contrary, the run of the algorithm and the construction of the underlying product automaton graph may interleave in such a way that new states of the product automaton are constructed *on-the-fly*, i.e. when they are needed by the algorithm. If this is the case, the algorithm may terminate due to the detection of an accepting cycle before the product automaton graph is fully constructed and all of its states are visited.

Those LTL model checking algorithms that may terminate before the state space is fully constructed are generally denoted as on-the-fly algorithms. If there is an error in the state space (accepting cycle), an on-the-fly algorithm may terminate in two possible phases: either an error is found before the interleaved generation of the product automaton graph is complete (i.e. before the algorithm detects that there are no new states to be explored), or an error is found after all states of the product automaton have been generated and the algorithm is aware of it. The first type of the termination is henceforward referred to as *early termination* (ET). Note that the awareness of completion of the product automaton construction procedure is important. If the algorithm detects the error by exploring the last state of the product automaton graph before it detects that it was actually the last unexplored state of the graph, we consider it to be an early termination.

We classify "on-the-flyness" of accepting cycle detection algorithms according to the capability of early termination as follows. An algorithm is

– *level 0 on-the-fly algorithm*, if there is a product automaton graph containing an error for which the algorithm will never early terminate.
– *level 1 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm may terminate early, but it is not guaranteed to do so.
– *level 2 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm is guaranteed to early terminate.

Note that level 0 algorithms are sometimes considered as on-the-fly algorithms and sometimes as non-on-the-fly algorithms depending on research community. Since a level 0 algorithm explores full state space of the product automaton graph it may be viewed as if it does not work on-the-fly. However, as explained above, just the fact that the algorithm employs product automaton construction is a good reason for considering the whole procedure of LTL model checking with a level 0 algorithm as an on-the-fly verification process.

To give examples of algorithms with appropriate classification we consider algorithms OWCTY, MAP, and Nested DFS. OWCTY algorithm is level 0 algorithm, MAP algorithm is level 1 algorithm and Nested DFS is level 2 algorithm. From the description in the previous section it is clear, that the algorithm we propose in this paper falls in the category of level 1.

It is not possible to give an analytical estimate of the percentage of the state space an on-the-fly algorithm needs to explore before early termination happens. Therefore, it is always important to accompany the classification of an algorithm by an experimental evaluation. This is in particular the case for level 1, where the experiments may give more accurate measure of the effectiveness of the method involved.

So far we have spoken only about the on-the-flyness status of a state space exploration algorithm. Nevertheless, *on-the-fly* LTL model checking procedure also denotes an approach that avoids explicit a priori construction of the negative claim automaton. We adapt the notation of [13] and denote this type of on-the-flyness as *truly* on-the-fly approach to LTL model checking. Note that truly on-the-flyness and algorithmic on-the-flyness are independent of each other and truly on-the-fly approach may be combined with on-the-fly algorithms of any level.

As for the state space exploration algorithms, the efficiency of the on-the-flyness of the algorithm may also be improved by other techniques. It might be the case that even the level 2 on-the-fly algorithm fails to discover an error, if the examined state space is large enough to exhaust system memory before an error is found. This issue has been addressed by methods of directed model checking [10,11,9], which combines model-checking with heuristic search. The heuristic guides the search process to quickly find a property violation so that the number of explored states is small. It is worthy to note that our approach can be extended with directed search as well.

## 5 Experiments

To experimentally evaluate efficiency of our approach we conducted numerous experiments employing models from BEEM [15]. All measured values were obtained using the verification tool `DiVinE-MC` version 1.4 [3,7]. The experiments were performed on a workstation equipped with two dual-core Intel Xeon 5130 @ 2.00 GHz processors, 16 GB of RAM, and 64-bit Linux-based operating system. For scalability experiments we also employed 16 way AMD Opteron 885 (8x dual-core) with 64 GB of RAM.

### 5.1 On-the-Flyness

For validation of the on-the-fly aspect of our new algorithm we originally selected 212 instances of verification problems with invalid LTL specification from BEEM database. However, we discovered that many of the instances resulted in a state space containing a self-loop over an accepting state (trivial accepting cycle). Such an accepting cycle can be easily detected using any graph traversal algorithm using just a simple self-loop test for each accepting state. After pruning out these unwanted cases, our benchmark contained 90 verification problems. An overview of the verification problems used to validate on-the-flyness of our approach is given in Figure 2.

We list experimental results in a few tables that all have a common structure. Each table row represents a single experimental configuration of the algorithm we run. Column *Algorithm* gives the configuration of the experiment. Columns *Visited states*, *Memory (MB)*, and *Time (s)* give the total number of distinct states generated, the total amount of memory consumed, and the total time of verification, respectively, for the whole benchmark set of verification problems. Column *ET ratio* reports on the number of *Early terminations* that happened for the experiment configuration. For example, if the *ET ratio* says 78/90, it means that for 78 verification problems out of 90, an accepting cycle was detected before the full state space was constructed.

| **Model** | **LTL Properties** | Validity |
|---|---|---|
| anderson | `G((!cs0) -> F cs0)` | No |
| driving_phils | `G(ac0 -> F gr0)` | No |
| | `GF ac0` | No |
| elevator2 | `G(r1->(F(p1 && co)))` | No |
| | `G(r1->(!p1U(p1U(p1&& co))))` | No |
| | `G(r1->(!p1U(p1U(!p1U(p1U(p1&&co))))))` | No |
| | `F(G p1)` | No |
| elevator | `G(waiting0 ->(F in_elevator0))` | No |
| iprotocol | `F consume` | No |
| | `G F consume` | No |
| | `((G F dataok) && (G F nakok)) -> (G F consume)` | No |
| lamport | `G (wait0 -> F (cs0) )` | No |
| | `G((!cs0) -> F cs0)` | No |
| lifts | `(GF pressedup0) -> (GF moveup)` | No |
| | `G (pressedup0 -> F moveup)` | No |
| | `((! moveup) U pressedup0) \|\| G (! moveup)` | No |
| mcs | `G (wait0 -> F (cs0) )` | No |
| | `G((!cs0) -> F cs0)` | No |
| peterson | `G (wait0 -> F (cs0) )` | No |
| | `G((!cs0) -> F cs0)` | No |
| | `GF someoneincs` | No |
| phils | `GF eat0` | No |
| | `G (one0 -> F eat0)` | No |
| | `GF someoneeats` | No |
| protocols | `(pready U prod0) -> ((cready U cons0) \|\| G cready)` | No |
| | `F (consume0 \|\| consume1)` | No |
| | `G F (consume0 \|\| consume1)` | No |
| rether | `G (res0 -> (rt0 R !cend))` | No |
| | `GF rt0` | No |
| | `G (want0 -> (! ce U (ce U (!ce && (rt0 R !ce)))))` | No |
| szymanski | `G (wait0 -> F (cs0) )` | No |
| | `G((!cs0) -> F cs0)` | No |
| | `GF someoneincs` | No |

**Fig. 2.** Selected BEEM models with invalid LTL properties

To identify the configuration of the algorithm in the experiment we use the following notation. $W = x$ denotes that the algorithm was performed using $x$ CPU cores (x workers in DiVinE-MC terminology), $V = y$ denotes that the algorithm involved $y$ different value propagations at the same time. Note that for $V = 0$ no values were propagated in order to early detect accepting cycles and the full state space of all verification problems had to be constructed. By *DFS* and *BFS* keys we distinguish whether the underlying search order employed for the initial reachability was a local depth-first or local breadth-first one, respectively. Also, since the behavior of the algorithm is non-deterministic (if more than one CPU cores are used) all values reported are actually average values obtained from ten independent runs of the corresponding experiment.

Before analyzing the experimental results, it is also important to explain the implementation of the technique we use to identify accepting states to be propagated. In particular, the algorithm always propagates the maximal accepting state it has encountered with respect to the given order of accepting states. To be able to efficiently decide about order of two given states, we decided not to compare the contents of the corresponding state vectors, but rather to use the unique pointers to memory addresses where the two state vectors are stored. For a state `s`, we denote the pointer by `ptr(s)`. Note that the ordering of states depends on properties of the memory managment system of the platform the program is running on. in practice, the ordering of states depends on the order in which the states were allocated, hence, on the order in which the states were examined. Some experiments employed multiple different orderings for identification of states to be propagated. Different orderings were achieved by performing various bit alternations in the bit representation of the pointer. Concrete techniques used in different configurations of our algorithm are listed in the following table.

| **Algorithm Configuration** | **Propagated values** | | |
|---|---|---|---|
| | 1st | 2nd | 3rd |
| V=0 | — | — | — |
| V=1 | `ptr(s)` | — | — |
| V=2 | `ptr(s)` | `ptr(s)` **xor** `0x555` | — |
| V=3 | `ptr(s)` | `ptr(s)` **xor** `0x555` | `ptr(s)` **xor** `0xFFFF` |

In Figure 3 we report results for single core experiments. It can be seen that the value propagation is quite successful regarding the early termination. Compared with the algorithm that performs no value propagation the algorithms with value propagations can save non-trivial amount of memory and reduce the runtime needed for verification, which definitely justifies our new algorithm to be considered as an algorithm that *works on-the-fly.* Other interesting aspect that can be read from the table are as follows. The more values are propagated, the larger is the ratio of early terminations, DFS mode seems to be slightly better in states and memory, but the BFS mode is better in detecting the presence of an

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| BFS, V=0, W=1 | 52 047 342 | 6 712 | 760 | 0/90 |
| BFS, V=1, W=1 | 23 157 474 | 4 858 | 295 | 66/90 |
| BFS, V=2, W=1 | 23 173 041 | 4 949 | 297 | 67/90 |
| BFS, V=3, W=1 | 20 175 952 | 4 796 | 237 | 78/90 |
| DFS, V=0, W=1 | 52 047 342 | 6 716 | 760 | 0/90 |
| DFS, V=1, W=1 | 19 849 655 | 4 583 | 272 | 56/90 |
| DFS, V=2, W=1 | 20 971 228 | 4 753 | 277 | 61/90 |
| DFS, V=3, W=1 | 17 090 024 | 4 502 | 240 | 68/90 |
| Nested DFS | 622 984 | 1 736 | 7 | 90/90 |

**Fig. 3.** Single core experiments

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| BFS, V=1, W=1 | 6 820 499 | 2 829 | 40 | 66/66 |
| BFS, V=2, W=1 | 6 854 458 | 2 893 | 41 | 67/67 |
| BFS, V=3, W=1 | 5 621 320 | 3 194 | 36 | 78/78 |
| DFS, V=1, W=1 | 3 930 520 | 2 257 | 23 | 56/56 |
| DFS, V=2, W=1 | 5 173 954 | 2 546 | 31 | 61/61 |
| DFS, V=3, W=1 | 1 802 949 | 2 518 | 12 | 68/68 |
| Nested DFS | 622 984 | 1 736 | 7 | 90/90 |

**Fig. 4.** Single core experiments restricted to runs with early termination

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| BFS, V=0, W=1 | 52 047 342 | 6 712 | 760 | 0/90 |
| BFS, V=0, W=2 | 52 047 342 | 9 072 | 503 | 0/90 |
| BFS, V=0, W=3 | 52 047 342 | 10 065 | 441 | 0/90 |
| BFS, V=0, W=4 | 52 047 342 | 10 874 | 395 | 0/90 |
| DFS, V=0, W=1 | 52 047 342 | 6 716 | 760 | 0/90 |
| DFS, V=0, W=2 | 52 047 342 | 9 069 | 504 | 0/90 |
| DFS, V=0, W=3 | 52 047 342 | 10 036 | 441 | 0/90 |
| DFS, V=0, W=4 | 52 047 342 | 10 888 | 396 | 0/90 |

**Fig. 5.** Experiments involving various number of CPU cores but no value propagation

| Algorithm | Visited states | Memory (MB) | Time (s) | ET ratio |
|---|---|---|---|---|
| BFS, V=1, W=1 | 23 157 474 | 4 858 | 295 | 66/90 |
| BFS, V=1, W=2 | 17 203 306 | 5 748 | 130 | 74/90 |
| BFS, V=1, W=3 | 20 244 429 | 6 955 | 122 | 74/90 |
| BFS, V=1, W=4 | 18 632 114 | 7 576 | 102 | 72/90 |
| DFS, V=1, W=1 | 19 849 655 | 4 583 | 272 | 56/90 |
| DFS, V=1, W=2 | 18 996 947 | 5 890 | 136 | 77/90 |
| DFS, V=1, W=3 | 22 826 318 | 7 037 | 138 | 73/90 |
| DFS, V=1, W=4 | 18 833 201 | 7 685 | 100 | 72/90 |

| Algorithm | Visited states | Memory (MB) | Time (s) | ET ratio |
|---|---|---|---|---|
| BFS, V=2, W=1 | 23 173 041 | 4 949 | 297 | 67/90 |
| BFS, V=2, W=2 | 17 540 622 | 5 976 | 132 | 75/90 |
| BFS, V=2, W=3 | 19 199 233 | 6 956 | 115 | 76/90 |
| BFS, V=2, W=4 | 18 856 858 | 7 647 | 102 | 73/90 |
| DFS, V=2, W=1 | 20 971 228 | 4 753 | 278 | 61/90 |
| DFS, V=2, W=2 | 18 557 211 | 5 909 | 136 | 76/90 |
| DFS, V=2, W=3 | 21 429 842 | 6 944 | 125 | 75/90 |
| DFS, V=2, W=4 | 18 601 625 | 7 712 | 98 | 72/90 |

| Algorithm | Visited states | Memory (MB) | Time (s) | ET ratio |
|---|---|---|---|---|
| BFS, V=3, W=1 | 20 175 952 | 4 796 | 237 | 78/90 |
| BFS, V=3, W=2 | 16 421 989 | 6 006 | 127 | 78/90 |
| BFS, V=3, W=3 | 17 335 622 | 6 765 | 108 | 80/90 |
| BFS, V=3, W=4 | 15 462 219 | 7 435 | 89 | 78/90 |
| DFS, V=3, W=1 | 17 090 024 | 4 502 | 240 | 68/90 |
| DFS, V=3, W=2 | 17 932 103 | 5 882 | 129 | 80/90 |
| DFS, V=3, W=3 | 21 174 728 | 6 984 | 126 | 76/90 |
| DFS, V=3, W=4 | 18 676 721 | 7 754 | 97 | 75/90 |

**Fig. 6.** Experiments involving various configurations of the algorithm and various number of CPU cores

accepting cycle on-the-fly. An interesting observation is the correspondence of the ratio of early terminations and the amount of visited states and time needed. For example, in *DFS, V=3, W=1* case, the ET ratio is 68/90 = 75%, the amount of avoided states is 35 millions which is 67% of the total of state spaces, and the time spared is 520 seconds, i.e. 72%. For comparison we also report the overall

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| exp 0, BFS, V=3, W=4 | 15 271 625 | 7 408 | 85 | 79/90 |
| exp 1, BFS, V=3, W=4 | 14 831 048 | 7 388 | 86 | 78/90 |
| exp 2, BFS, V=3, W=4 | 16 324 239 | 7 541 | 90 | 78/90 |
| exp 3, BFS, V=3, W=4 | 14 979 049 | 7 400 | 91 | 78/90 |
| exp 4, BFS, V=3, W=4 | 16 064 605 | 7 453 | 90 | 77/90 |
| exp 5, BFS, V=3, W=4 | 15 950 789 | 7 445 | 87 | 80/90 |
| exp 6, BFS, V=3, W=4 | 14 726 197 | 7 401 | 85 | 79/90 |
| exp 7, BFS, V=3, W=4 | 15 601 260 | 7 441 | 94 | 78/90 |
| exp 8, BFS, V=3, W=4 | 15 308 205 | 7 413 | 90 | 79/90 |
| exp 9, BFS, V=3, W=4 | 15 565 178 | 7 462 | 90 | 75/90 |
| **Maximum** | 16 324 239 | 7 541 | 94 | 80/90 |
| **Minimum** | 14 726 197 | 7 388 | 85 | 75/90 |
| **Average** | 15 462 220 | 7 435 | 88.8 | 78.1/90 |

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| exp 0, DFS, V=3, W=4 | 19 126 324 | 7 802 | 98 | 75/90 |
| exp 1, DFS, V=3, W=4 | 17 513 441 | 7 622 | 101 | 75/90 |
| exp 2, DFS, V=3, W=4 | 19 289 379 | 7 814 | 98 | 73/90 |
| exp 3, DFS, V=3, W=4 | 18 234 139 | 7 734 | 97 | 73/90 |
| exp 4, DFS, V=3, W=4 | 16 135 286 | 7 504 | 87 | 78/90 |
| exp 5, DFS, V=3, W=4 | 19 586 932 | 7 833 | 98 | 74/90 |
| exp 6, DFS, V=3, W=4 | 19 237 964 | 7 803 | 94 | 78/90 |
| exp 7, DFS, V=3, W=4 | 20 121 416 | 7 885 | 105 | 74/90 |
| exp 8, DFS, V=3, W=4 | 18 956 781 | 7 784 | 93 | 78/90 |
| exp 9, DFS, V=3, W=4 | 18 565 549 | 7 767 | 97 | 75/90 |
| **Maximum** | 20 121 416 | 7 885 | 105 | 78/90 |
| **Minimum** | 16 135 286 | 7 504 | 87 | 73/90 |
| **Average** | 18 676 721 | 7 754 | 96.8 | 75.3/90 |

| **Algorithm** | **Visited states** | **Memory (MB)** | **Time (s)** | **ET ratio** |
|---|---|---|---|---|
| BFS, V=3, W=4 | 15 462 220 | 7 435 | 88.8 | 78.1/90 |
| DFS, V=3, W=4 | 18 676 721 | 7 754 | 96.8 | 75.3/90 |

**Fig. 7.** Non-deterministic behavior of the algorithm demonstrated on version V=3 and 4 CPU cores. Comparison of BFS and DFS search order strategies.

| **Model** | **LTL Properties** | **Validity** |
|---|---|---|
| anderson | `GF someoneincs` | Yes |
| elevator2 | `G(r0->(!p0U(p0U(!p0U(p0U(p0&&co))))))` | Yes |
| lamport | `GF someoneincs` | Yes |
| leader_filters | `F leader` | Yes |
| rether | `GF (nact0)` | Yes |
| szymanski | `GF someoneincs` | Yes |

**Fig. 8.** Selected BEEM model instances with valid LTL properties

| | `16-way AMD Opteron` | | | | | `4-way Intel Xeon` | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Model** | 1 | 2 | 4 | 8 | 16 | 1 | 2 | 3 | 4 |
| `anderson` | 11:41 | 6:27 | 4:46 | 2:45 | 1:56 | 9:39 | 5:23 | 4:10 | 3:56 |
| `elevator2` | 9:27 | 5:40 | 3:28 | 2:07 | 1:35 | 8:18 | 4:51 | 3:50 | 3:05 |
| `lamport` | 23:12 | 13:23 | 8:10 | 5:16 | 3:39 | 19:41 | 10:58 | 8:28 | 6:49 |
| `leader_filters` | 9:04 | 5:10 | 3:08 | 2:25 | 1:45 | 7:34 | 5:02 | 3:23 | 2:53 |
| `rether` | 2:22 | 1:12 | 58 | 38 | 27 | 2:06 | 1:05 | 58 | 55 |
| `szymanski` | 1:20 | 51 | 39 | 33 | 28 | 1:09 | 43 | 39 | 35 |

**Fig. 9.** Scalability experimental results of liveness checking on a selection of models with valid properties

values of visited states and time needed if the serial Nested DFS algorithm is used.

Figure 4 gives the overall values if only the cases, where early termination happened, are considered. The table demonstrates, that if early termination succeeds, the efficiency of our new algorithm is quite close to the optimal but serial Nested DFS algorithm. Note the increase in the number of visited states in case *DFS, V=2, W=1* compared to *DFS, V=1, W=1*. We explain this by the fact, that in the case of *V=2* the memory requirements to store a single state vector differs from the case *V=1*, hence, pointers to addresses of state vectors are reordered due to the underlying memory management.

Before we discuss how the algorithm performs with respect to early termination if multiple CPU cores are used, we first look into how the algorithm behaves if no value propagation is used. As it can be seen from Figure 5, using more CPU cores not only renders shorter running times, but it also increases the overall memory consumption. This can be easily explained by the overhead related to multiple threads. For example, in `DiVinE-MC` every thread maintains its own hash table. However, there is an interesting phenomenon, also independent of the search order used, that the increase from one core to two cores is approximately twice as big as any further increase from n cores to n+1 cores. Our guess is that for a single core run, the tool consumes less memory as the

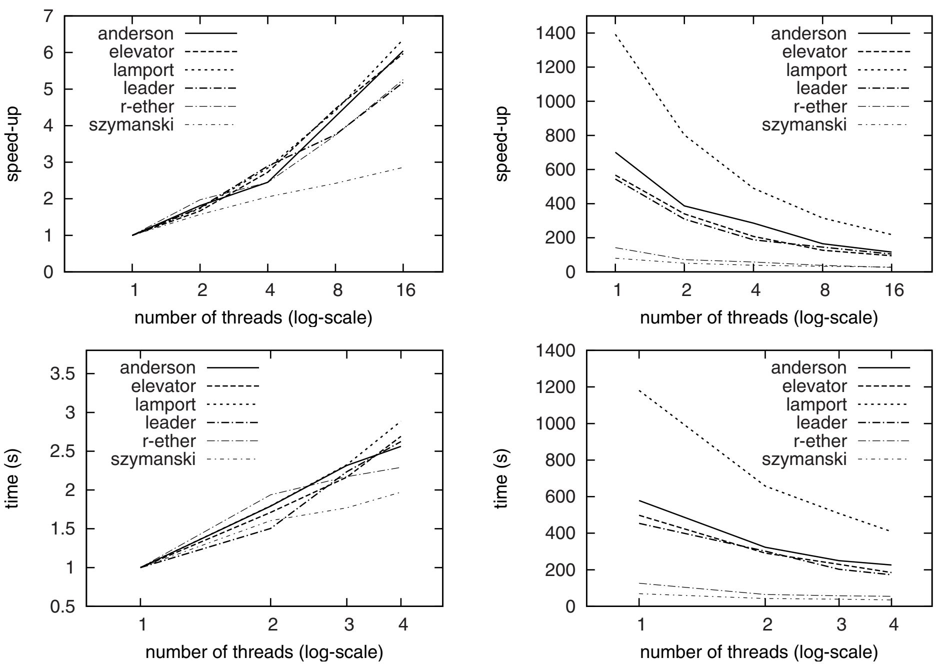


**Fig. 10.** Runtimes and speedup plots as measured on 16-way AMD Opteron 885 and 4-way Intel Xeon platforms

underlying memory management need not pre-allocate large memory blocks to prevent fragmentation.

In Figure 6 we present an overview of our experimental study. We conclude from the experimental results that our parallel algorithm for accepting cycle detection works in an on-the-fly manner. The experimental data demonstrate that using more accepting states for the propagation increases the successfulness of early termination, though it is disputable whether it actually reduces demands on computing resources. An interesting point is that unlike the single core case, in parallel processing *BFS* variants outperform *DFS* ones. This result is however, bound to the ordering of states in the state space.

Finally, data in Figure 5 demonstrate the non-deterministic behavior of parallel runs. It can be observed that the early termination ratio and the demands on computational resources vary, however, the deviation is relatively small which is very important from the practical point of view.

## 5.2 Scalability

In order to demonstrate the scalability aspects of the new algorithm we selected various valid instances from the BEEM database. See Figure 8 for details. In Figure 9 we report on run-times needed to complete the corresponding verification tasks. It can be seen that the efficiency of parallel computation is slightly deteriorating as the number of cores involved in the computation reaches the maximum number of available cores. Nevertheless, the run-times consistently decrease as the number of cores involved increases. The speedup and run-times are also given as graphs in Figure 10.

## 6 Conclusions

In this paper we described a new parallel algorithm for accepting cycle detection problem, i.e. explicit-state LTL model-checking. The algorithm emerged as a combination of two existing parallel algorithms, OWCTY and MAP, keeping the best of both. In particular, the new parallel algorithm is scalable and time-optimal for majority of LTL properties, likewise the OWCTY algorithm, but it is also able to detect some accepting cycles on-the-fly, likewise the MAP algorithm. No such algorithm has been known so far.

We also performed large experimental study. It demonstrated that using our new algorithm significantly reduces computation resources needed to complete the verification task in many cases.

As for the future work, we can see many options. First of all, we have the impression that one could further improve the results by clever selection of ordering function. It is clear that technique to select states to be propagated influences the experimental results a lot. It is still unclear how far one can get with a good ordering function in practice. Another future goal is to incorporate directed search in the Initialize phase of the algorithm. Directed search is known to significantly increase efficiency of early termination in serial case, we expect this to be the case also for parallel algorithms. And finally, we still do not have the answer to the open problem of existence of parallel scalable and optimal level 2 on-the-fly algorithm for weak LTL properties and level 1 or better for full LTL.

## References

1. Barnat, J., Brim, L., Černá, I.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
3. Barnat, J., Brim, L., Ročkai, P.: DiVinE Multi-Core – A Parallel LTL Model-Checker. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
4. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
5. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
6. Courcoubetics, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)

7. DiVinE – Distributed Verification Environment, Masaryk University Brno, `http://divine.fi.muni.cz`
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-State Verification. In: Proc. Workshop on Formal Methods in Software Practice, pp. 7–15. ACM Press, New York (1998)
9. Edelkamp, S., Jabbar, S.: Large-scale directed model checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
10. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2-3), 247–267 (2004)
11. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed Explicit Model Checking with HSF-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
12. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. 345(1), 60–82 (2005)
13. Hammer, M., Knapp, A., Merz, S.: Truly On-the-Fly LTL Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 191–205. Springer, Heidelberg (2005)
14. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In: CAV 2009. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)
15. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
16. Schwoon, S., Esparza, J.: A Note on On-The-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
17. Tarjan, R.: Depth First Search and Linear Graph Algorithms. SIAM Journal on Computing, 146–160 (January 1972)
18. Vardi, M.Y.: Automata-Theoretic Model Checking Revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 137–150. Springer, Heidelberg (2007)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. IEEE Symposium on Logic in Computer Science, pp. 322–331. Computer Society Press (1986)

2009 15th International Conference on Parallel and Distributed Systems

# CUDA accelerated LTL Model Checking

Jiří Barnat, Luboš Brim, Milan Češka, and Tomáš Lamr
*Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic*

## Abstract

*Recent technological developments made available various many-core hardware platforms. For example, a SIMD-like hardware architecture became easily accessible for many users who have their computers equipped with modern NVIDIA GPU cards with CUDA technology. In this paper we redesign the maximal accepting predecessors algorithm [7] for LTL model checking in terms of matrix-vector product in order to accelerate LTL model checking on many-core GPU platforms. Our experiments demonstrate that using the NVIDIA CUDA technology results in a significant speedup of verification process.*

## 1. Introduction

Model-checking [1] is a wide-spread technique for automated formal verification. Given a formal description of a system and desired system property, the goal of the model-checking procedure is to analyze reachable system configurations in order to decide whether the system satisfies the property or not. In this paper we deal with LTL model checking, which is the case when the property to be verified is given as a formula of Linear Temporal ogic (LTL). In LTL model checking, the question of satisfaction of the property can be reduced to the problem of detection of an accepting cycle (cycle through at least one vertex denoted as accepting vertex) in a directed graph.

All the know algorithms for accepting cycle detection can be divided into two classes. Algorithms such as Nested DFS [10] or algorithms based on Tarjan's SCC decomposition algorithm [17], [18] exhibit optimal (linear) time complexity, but are incompatible with parallel processing. This is because they strongly rely on the so called depth-first search (DFS) postorder for computation of which no scalable parallel algorithm is known [16]. The other group of algorithms for accepting cycle detection are algorithms such as OWCTY [9], [12] or MAP [7] that avoid DFS postorder, but exhibit unoptimal time complexity. However, it has been demonstrated that the unoptimality is easily outweighted by parallel processing [2], [19]. As a result, the unoptimal algorithms are actually faster than the optimal sequential algorithms if contemporary parallel hardware is used.

*This work has been supported in part by the Czech Grant Agency grants No. 201/09/P497, 201/09/1389, 102/09/H042 and the Academy of Sciences grant No. 1ET408050503.*

Moreover, the graph to be analyzed tends to be very large for realistic systems and it is handled only with difficulties by a single memory-limited machine. Consequently, optimal utilization of resources of various hardware platforms have got much attention by the model checking community. As most modern hardware platforms are actually parallel platforms, the desire for full utilization of the power available rendered all sequential algorithms obsolete.

Modern graphics processing units (GPUs) have emerged as a revolutionary technological opportunity due to their tremendous massive parallelism, floating point capability, low cost, and ubiquitous presence in commodity computer systems. Many key computational kernels have been redesigned to exploit the performance of this modern hardware. The key to effective utilization of GPUs for scientific computing is the design and implementation of efficient data-parallel algorithms that can scale to hundreds of tightly coupled processing units.

In this paper we show how one of the parallel algorithms for accepting cycle detection, namely the MAP algorithm, can be effectively accelerated on GPU if the input data are given in an appropriate format. The MAP algorithm was chosen, because, unlike the OWCTY algorithm, it allows on-the-fly verification.

The rest of the paper is organized as follows. In Section 2 we briefly recall basics of automata-theoretic approach to LTL Model Checking. Sections 3, 5 and 6 describe how we adapted the algorithm MAP to GPU processing and what enhancements we did to achieve good performance. In Sections 4 we recapitulate NVIDIA CUDA hardware platform a show how our adapted algorithm can be implemented on CUDA. Section 7 reports on an experimental evaluation of our approach, and finally, Section 8 summarizes achieved results and plots some future directions.

## 2. LTL Model Checking

For LTL model checking purposes, the system to be analyzed has to be described in some modeling language, `ProMeLa` [14] for example, and the property to be checked has to be given as formula of Linear Temporal Logic (LTL) [1]. To answer the LTL model checking question, tools, such as `SPIN` [14] or `DiVinE` [5], employ automata-theoretic approach to reduce the model checking problem to the problem of non-emptiness of Büchi automata. In particular, the model of a system $S$ is viewed as a finite automaton $A_S$ describing all possible behaviors of the system. The property to be checked (LTL formula $\varphi$) is negated and translated into Büchi automaton $A_{\neg\varphi}$ describing all the behaviors violating $\varphi$. In order to check whether the system violates $\varphi$, a synchronous product $A_S \times A_{\neg\varphi}$ of $A_S$ and $A_{\neg\varphi}$ is constructed describing those behaviors of the system that violates $\varphi$, i.e. $L(A_S \times A_{\neg\varphi}) = L(A_s) \cap L(A_{\neg\varphi})$. The automata $A_S$, $A_{\neg\varphi}$, and $A_S \times A_{\neg\varphi}$ are referred to as *system*, *property*, and *product* automata, respectively. System $S$ satisfies formula $\varphi$ if and only if the language of the product automaton is empty, which is if and only if there is no reachable accepting cycle in the underlying graph of the product automaton. The LTL model checking problem is thus reduced to the problem of the detection of an accepting cycle in the product automaton graph.

There are several parallel algorithms for accepting cycle detection. One of them is the algorithm MAP [7] which we now briefly introduce in its *successor* version. Let $G = (V, E, v_0, \mathcal{A})$ be the graph of the product automaton, where $V$ is a finite set of vertices, $E$ is a set of edges, $v_0$ is an initial vertex, and $\mathcal{A}$ is a vertex predicate indicating whether a state is accepting or not. Let $<$ be a linear ordering of the set of vertices, given e.g. by the vertex numbering. We extend the ordering to the set $V \cup \{\perp\}$ ($\perp \notin V$) and put $\perp < v$ for all $v \in V$. Furthermore, let $map :: V \rightarrow V \cup \{\perp\}$ is a function returning the maximal accepting successor of a given vertex or $\perp$ if it does not exist, i.e. $map(u) = \max\{\perp, v \mid (u,v) \in E^+ \wedge \mathcal{A}(v)\}$.

The idea of the algorithm to detect an accepting cycle is as follows. If a vertex $u$ is its own maximal accepting successor, i.e. $u = map(u)$, the presence of an accepting cycle is guaranteed. If there is an accepting cycle in the graph, but for none of its vertices $u = map(u)$, then the maximal accepting successor of all the vertices of the cycle must be the same, must lie outside the cycle and can thus be marked as non-accepting. The idea of the algorithm is to process the graph in a few iterations so that each iteration computes map values for all the vertices. If no accepting cycle is discovered, all maximal accepting successors that occur in $map(u)$ for some $u$ are marked as non-accepting for all the following iterations. The algorithm iterates until an accepting cycle is found or the set of accepting vertices becomes empty. See the pseudo-code in Algorithm 1. A key procedure of the algorithm is COMPUTEALLMAPS() that is responsible for computing the values of the function *map* for all the vertices reachable from the initial vertex. See the pseudo-code in Algorithm 2. Initially, the values of $map(u)$ are set to $\perp$ for all $u \in V$. These values are then repeatedly updated until a global fix-point is reached, i.e. no update can be done for any value of $map(u)$. Suppose a directed edge $(u,v)$ from $u$ to $v$, the new value of $map(u)$, the so called *update* along the edge $(u,v)$, is computed using function $maxacc(u,v)$ as follows:

**Algorithm 1** Algorithm MAP

```
Input:  directed graph $G = (V, E, v_0, \mathcal{A})$ of $A_{S\times\neg\varphi}$
        linear ordering $<$ on $V$
Output: true,   if $A_{S\times\neg\varphi}$ contains accepting cycle
        false,  otherwise

1: while ($\exists v \in V : \mathcal{A}(v) = $ true) do
2:    COMPUTEALLMAPS(G,<)
3:    if ($\exists u \in V : u = map(u)$) then
4:       return true
5:    end if
6:    $\mathcal{A}(v) \leftarrow$ false
7: end while
8: return false
```

$$maxacc(u,v) = \begin{cases} \max\{map(u), map(v), v\} & \text{if } \mathcal{A}(v) \\ \max\{map(u), map(v)\} & \text{otherwise.} \end{cases}$$

**Algorithm 2** COMPUTEALLMAPS(G,<)

```
Input:  directed graph $G = (V, E, v_0, \mathcal{A})$
        linear ordering $<$ on $V$
Output: value of $map(u)$ for all $u \in V$

1: for all ($u \in V$) do
2:    $map(u) = \perp$
3: end for
4: while ($\neg$ fix-point) do
5:    for all ($(u,v) \in E$) do
6:       if ($v \in \mathcal{A}$) then
7:          $map(u) \leftarrow \max\{map(u), map(v), v\}$
8:       else
9:          $map(u) \leftarrow \max\{map(u), map(v)\}$
10:      end if
11:   end for
12: end while
13: return map
```

Henceforward, we also refer to the iterations of the while loop of Algorithm MAP given in Algorithm 1 as of *outer* iterations, and the iterations of the while loop of procedure COMPUTEALLMAPS given in Algorithm 2 as of *inner* iterations.

The practical performance of the basic algorithm may be further enhanced if the graph to be checked for the presence of an accepting cycle is partitioned into subgraphs so that no cycle of the original graph maps to multiple partitions. In that case the *inner* iterations as performed in procedure COMPUTEALLMAPS may be prevented from propagating values of *map* along edges that cross partition boundaries. This brings no complexity improvement, but it generally reduces the number of inner iterations needed to achieve the fix-point.

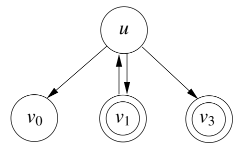
One technique to partition the product automaton graph is part of the algorithm itself. It builds upon the fact that if two vertices differ in their values of *map*, they cannot lie on the same cycle. Therefore, the propagation in procedure COMPUTEALLMAPS may be localized to those edges $(u,v)$ for which the values of $map(v)$ and $map(u)$ computed in the previous outer iteration are the same. The values of *map* function from the previous outer iteration are referred to as *oldmap* values.

## 3. Reformulation of MAP algorithm

In order to accelerate the MAP algorithm on CUDA we reformulate it as a matrix-vector multiplication algorithm.

Let us assume the graph $G$ of the product automaton $A_{S\times\neg\varphi}$ is represented as an adjacency matrix $M$. The matrix keeps value 1 at row $u$ and column $v$ for every directed edge $(u,v)$. See Figure 1. Additional data to be stored with every vertex of the graph are not stored directly in the matrix, but they are rather organized in separate vectors. Namely, vector $\vec{m}$ of *map* values, vector $\vec{o}$ of *oldmap* values, vector $\vec{\mathcal{A}}$ of values of the predicate $\mathcal{A}$, and an output vector $\vec{r}$ of bits indicating a recent update to the value of *map*. Vector $\vec{r}$ is used to detect the fix-point of computation of inner iterations, i.e. the situation when no update to *map* function occurs in two successive inner iterations.

The algorithm proceeds as illustrated in Figure 1. Initially, all *map* and *oldmap* values are set to $\bot$, see the column vectors $\vec{m}$ and $\vec{o}$. The algorithm repeatedly updates values of the vector $\vec{m}$ until a fix-point is reached (the three topmost matrix-vector product equations). Since $map(v) \neq v$ for all vertices $v$, the maximal accepting vertex $v_3$ cannot be part of an accepting cycle ($map(v_3) < v_3$). The algorithm resets its accepting status, copies *map* values to *oldmap* values and sets all values of *map* to $\bot$. Note that there are two subgraphs to be further processed identified



$$\begin{array}{cc} & \begin{array}{cccc} v_0 & v_1 & & v_3 \end{array} \\ u & \left(\begin{array}{cccc} 0&0&0&0\\ 0&0&1&0\\ 1&1&0&1\\ \hline 0&0&0&0 \end{array}\right) \end{array} \times \begin{array}{ccc} \vec{m} & \vec{o} & \vec{\mathcal{A}} \\ \bot&\bot&0\\ \bot&\bot&1\\ \bot&\bot&0\\ \bot&\bot&1 \end{array} = \begin{array}{cccc} \vec{m} & \vec{o} & \vec{\mathcal{A}} & \vec{r} \\ \bot&\bot&0&0\\ \bot&\bot&1&0\\ 3&\bot&0&1\\ \hline \bot&\bot&1&0 \end{array}$$

$$v_1 \left(\begin{array}{cccc} 0&0&0&0\\ 0&0&1&0\\ \hline 1&1&0&1\\ 0&0&0&0 \end{array}\right) \times \begin{array}{ccc} \bot&\bot&0\\ \bot&\bot&1\\ 3&\bot&0\\ \bot&\bot&1 \end{array} = \begin{array}{cccc} \bot&\bot&0&0\\ 3&\bot&1&1\\ \hline 3&\bot&0&0\\ \bot&\bot&1&0 \end{array}$$

$$\left(\begin{array}{cccc} 0&0&0&0\\ 0&0&1&0\\ 1&1&0&1\\ 0&0&0&0 \end{array}\right) \times \begin{array}{ccc} \bot&\bot&0\\ 3&\bot&1\\ 3&\bot&0\\ \bot&\bot&1 \end{array} = \begin{array}{cccc} \bot&\bot&0&0\\ 3&\bot&1&0\\ 3&\bot&0&0\\ \bot&\bot&1&0 \end{array}$$

$$\begin{array}{ccc} m & o & \mathcal{A} \\ \bot&\bot&0\\ 3&\bot&1\\ 3&\bot&0\\ \bot&\bot&1 \end{array} \Longrightarrow \begin{array}{ccc} m & o & \mathcal{A} \\ \bot&\bot&0\\ 3&\bot&1\\ 3&\bot&0\\ \bot&3&0 \end{array} \Longrightarrow \begin{array}{ccc} m & o & \mathcal{A} \\ \bot&\bot&0\\ \bot&3&1\\ \bot&3&0\\ \bot&3&0 \end{array}$$

$$\left(\begin{array}{cccc} 0&0&0&0\\ 0&0&1&0\\ 1&1&0&1\\ \hline 0&0&0&0 \end{array}\right) \times \begin{array}{ccc} \bot&\bot&0\\ \bot&3&1\\ \bot&3&0\\ \bot&3&0 \end{array} = \begin{array}{cccc} \bot&\bot&0&0\\ \bot&3&1&1\\ 1&3&0&0\\ \hline \bot&3&0&0 \end{array}$$

$$\left(\begin{array}{cccc} 0&0&0&0\\ 0&0&1&0\\ \hline 1&1&0&1\\ 0&0&0&0 \end{array}\right) \times \begin{array}{ccc} \bot&\bot&0\\ \bot&3&1\\ 1&3&0\\ \bot&3&0 \end{array} = \begin{array}{cccc} \bot&\bot&0&0\\ 1&3&1&0\\ \hline 1&3&0&0\\ \bot&3&0&0 \end{array}$$

Figure 1. Matrix vector computation of MAP.

by the *oldmap* values. Vertex $v_3$ is part of one of the subgraphs, but since it is not accepting anymore, it cannot influence any future values of *map* computed for vertices within the subgraph. Then the next outer iteration proceeds. The algorithm detects (using two inner iterations) that $map(v_1) = v_1$ and so it terminates reporting accepting cycle through vertex $v_1$.

The key observation is that the vector of *map* values computed in each inner iteration is computed as a matrix-vector product by substituting `max` for the standard $+$ operation, and `maxacc` for the standard $\cdot$ operation. The result, however, considers only those summands for which *oldmap* values are the same as *oldmap* value of the updated vertex. For example, the resulting value of $\vec{m}[u]$ is computed as

$$\vec{m}[u] = max_{1<i\leq 4}\ (M[u][i] \cdot old \cdot \texttt{maxacc(i,u)})$$

where $old$ equals to 1 if $\vec{o}[u] = \vec{o}[i]$ and equals to 0 in the other case. Also note that values of $\vec{\mathcal{A}}$ are accessed within `maxacc` operation and that 0 encodes $\bot$.

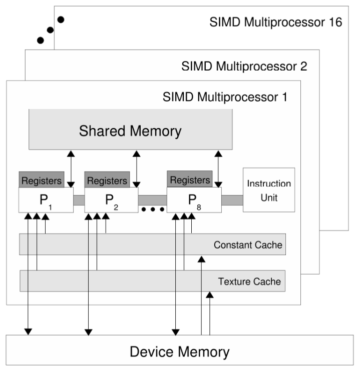


Figure 2. CUDA hardware model

## 4. CUDA Architecture

The Compute Unified Device Architectures (CUDA) [11], developed by NVIDIA, is parallel programing model and software environment providing general purpose programming on Graphics Processing Units. At the hardware level, GPU device is a collection of multiprocessors each consisting of eight scalar processor cores, instruction unit, on-chip shared memory, and texture and constant memory caches. Every core has a large set of local 32-bit registers but no cache. See the structure as depicted in Figure 2. The multiprocessors follow the SIMD architecture, i.e. they concurrently execute the same program instruction on different data. Communication among multiprocessors is realized through the shared device memory that is accessible for every processor core.

On the software side, the CUDA programming model extends the standard C/C++ programming language with a set of parallel programming supporting primitives. A CUDA program consist of a *host* code running on a CPU and a *device* code running on the GPU. The device code is structured into so called *kernels*. A kernel executes the same scalar sequential program in many *data independent parallel threads*. Within the kernel, threads are organized into thread blocks forming a grid of one or more blocks, see Figure 3. Each thread is given a unique index within its block *threadIdx* and each block is given a unique index *blockIdx* within the grid. The threads of a single block are guaranteed to be executed on the same multiprocessor, thus, they can easily access data stored in shared memory of the multiprocessor. The programmer specifies both the number of blocks and number of threads per block to be created before a kernel is launched. These values are available to the kernel as *gridDim* and *blockDim* values, respectively.

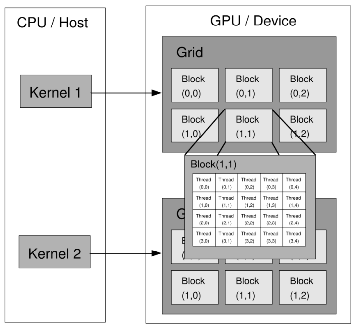


Figure 3. CUDA programming model

$$M = \begin{pmatrix} a & b & c & 0 \\ 0 & d & e & 0 \\ 0 & 0 & 0 & 0 \\ f & 0 & g & h \end{pmatrix} \qquad \begin{array}{l} \qquad\qquad\text{Row 0} \quad \text{Row 1} \quad \text{Row 3} \\ M_r[8] = \{\ a\ b\ c\ d\ \ e\ \ f\ g\ h\ \} \\ M_c[8] = \{\ 0\ 1\ 2\ 1\ \ 2\ \ 0\ 2\ 3\ \} \\ \hline M_n[4] = \{\ 0\ 3\ 5\ 8\ \} \end{array}$$

Figure 4. A sparse matrix and its CSR representation.

Using CUDA to accelerate the computation is easily exemplified on a vector summation problem. Suppose two vectors of length $n$ to be summed. In the standard imperative programming language, a programmer would use a for loop to sum individual vector elements successively. Using CUDA, however, the vector elements can be summed concurrently in a single *kernel* call populated with $n$ threads, each responsible for summation of a single pair of vector elements at the position given be the thread index.

## 5. CUDA Accelerated Algorithm MAP

Data structures used for CUDA accelerated computation must be designed with care. First, they have to allow independent thread-local data processing so that the CUDA hardware can employ massive parallelism. And second, they have to be small so that the high latency device-memory access and limited device-memory bandwidth are not large performance bottlenecks. As for the algorithm MAP, it is the matrix representation of the graph $A_{S\times\neg\varphi}$ to be encoded appropriately at the first place. Note that uncompressed matrix or dynamically linked adjacency lists violate the requirements and as such they are inappropriate for CUDA computing.

We decided to encode the matrix of the product automaton graph as a sparse matrix using *compresse sparse row* (CSR) format. In this format a sparse matrix is encoded using three one-dimensional arrays $M_r$, $M_c$, and $M_n$ as follows. All the non-zero elements of a matrix $M$ are stored in the array $M_r$ in left-to-

**Algorithm 3** CUDA MAP Algorithm - host code

**Input:** directed graph $G = (V, E, v_0, \mathcal{A})$ of $A_{S\times\neg\varphi}$
**Output:** `true`, if $A_{S\times\neg\varphi}$ contains accepting cycle
`false`, otherwise

```
1:  CREATE_CSR_REPRESENTATION($G, M_c, M_n, \vec{m}$)
2:  $acc\_cycle\_found \leftarrow$ false
3:  $repropagate \leftarrow$ true
4:  $unmarked \leftarrow$ true
5:  copy $(M_c, M_n, \vec{m})$ to GPU $(gM_c, gM_n, \vec{gm})$
6:  while $unmarked \wedge \neg acc\_cycle\_found$ do
7:      while $repropagate \wedge \neg acc\_cycle\_found$ do
8:          $repropagate \leftarrow$ false
9:          $map\_Kernel(gM_c, gM_n, \vec{gm}, acc\_cycle\_found)$
10:         $check\_repropagate\_Kernel(\vec{gm}, repropagate)$
11:     end while
12:     $unmarked \leftarrow$ false
13:     $unmark\_acc\_vertices\_Kernel(\vec{gm}, unmarked)$
14: end while
15: return $acc\_cycle\_found$
```

**Algorithm 4** device code - *map_Kernel*

```
proc $map\_Kernel(gM_c, gM_n, \vec{gm}, acc\_cycle\_found)$
1:  $row \leftarrow blockIdx * blockDim + threadIdx$
2:  if $row < |\vec{gm}|$ then
3:      $row\_begin \leftarrow gM_n[row]$
4:      $row\_end \leftarrow gM_n[row + 1]$
5:      $u \leftarrow gm[row]$
6:      $propagate \leftarrow \perp$
7:      $u.updated \leftarrow$ false
8:      for $column \leftarrow row\_begin$ to $row\_end - 1$ do
9:          $v \leftarrow gm[gM_c[column]]$
10:         if $u.map = v.old \wedge u.old \neq v.old$ then
11:             $u.old \leftarrow v.old$
12:             $u.map \leftarrow \perp$
13:             $propagate \leftarrow \perp$
14:             $u.repropagate \leftarrow$ true
15:             break
16:         else if $u.old = v.old$ then
17:             $propagate \leftarrow max(propagate, maxacc(u, v))$
18:         end if
19:     end for
20:     if $propagate = row$ then
21:         $acc\_cycle\_found \leftarrow$ true
22:     end if
23:     if $propagate > u.map$ then
24:         $u.map \leftarrow propagate$
25:         $u.repropagate \leftarrow$ true
26:     end if
27:     $gm[row] \leftarrow u$
28: end if
```

right and top-to-bottom order. The array $M_c$ keeps the corresponding column indices for every element in $M_r$, while the array $M_n$ keeps positions of first elements of rows of $M$ in arrays $M_n$ and $M_r$. See Figure 4. Note that in our case all the non-empty elements of the matrix are the same, hence, the array $M_r$ is redundant for our purposes and we do not maintain it at all.

The other data structures are organized as vectors, which is compatible with CUDA processing. The values of *map*, *old map*, $\mathcal{A}$ predicate, and repropagation bit $r$ for vertex $i$ are available in the pseudo-code as $m[i].map$, $m[i].old$, $m[i].acc$ and $m[i].repropagate$, respectively. Since the values of *map* and *oldmap* are technically pointers, we were able to store the two other bits of information into unused pointer bits reducing thus the space needed to record all the data for one vertex to two times 4 Bytes.

As explained in Section 3, the major computation part of the algorithm MAP can be formulated in terms of matrix-vector product. Given CSR matrix representation and a column vector, an efficient CUDA accelerated matrix-vector product procedure was described in [6], [13]. The idea of the procedure is to map every row of the matrix to one thread. Since in our case the edges of the graph are more or less uniformly spread in the matrix, this approach leads to a satisfactory balanced load of CUDA cores.

The pseudo-code of the CUDA accelerated algorithm MAP follows. Algorithm 3 lists the overall host code, i.e. the part that is executed on the CPU. The inner and outer while loops listed in the pseudo-code correspond with the inner and outer iterations as introduced in Section 2.

There are three kernel functions called from the host code. The most important one, *map_Kernel*, is listed as Algorithm 4. Every call to *map_Kernel* performs one matrix-vector product operation, i.e. it propagates the *map* values once along every edge (see lines 17 and 23-26 of Algorithm 4). Note however, that the very first call to the kernel in every outer loop does a slightly different job. In particular, it copies *map* values to *oldmap* values to decompose the graph according to *map* values from the previous outer iteration (see lines 10-15 of Algorithm 4). If no accepting cycle is found and *map_Kernel* returns, *check_repropagate_Kernel* is called to detect a fixpoint. *check_repropagate_Kernel* is listed as Algorithm 5. If there is no *map* value to be further propagated, the outer iteration is completed by a call to *unmark_acc_vertices_Kernel* to unset accepting predicate for accepting states proven to be outside an accepting cycle. *unmark_acc_vertices_Kernel* is listed as Algorithm 6.

Note that an update of *map* value of a single vertex requires access to all immediate successors of it, which can be done effectively in CSR matrix representation. If we opted for the predecessor version of the algorithm, the algorithm would require to access

**Algorithm 5** device code - *check_repropagate_Kernel*

```
proc $check\_repropagate\_Kernel(\vec{gm}, repropagate)$
1:  $row \leftarrow blockIdx * blockDim + threadIdx$
2:  if $row < |\vec{gm}|$ then
3:      $u \leftarrow gm[row]$
4:      if $u.repropagate$ then
5:          $repropagate \leftarrow$ true
6:      end if
7:  end if
```

**Algorithm 6** dev.code - *unmark_acc_vertices_Kernel*

```
proc $unmark\_acc\_vertices\_Kernel(\vec{gm}, unmarked)$
1:  $row \leftarrow blockIdx * blockDim + threadIdx$
2:  if $row < |\vec{gm}|$ then
3:      $u \leftarrow gm[row]$
4:      if $u.acc \wedge u.map < row$ then
5:          $u.map \leftarrow \perp$
6:          $u.old \leftarrow row$
7:          $u.acc \leftarrow$ false
8:          $gm[row] \leftarrow u$
9:          $unmarked \leftarrow$ true
10:     end if
11: end if
```

immediate predecessors of vertices, which would mean to transpose the matrix first. This would prevent on-the-fly computation at all.

## 6. On-The-Fly Verification

The last not-yet-discussed but quite essential procedure of the whole verification process is the transformation of the input data as given to the model checker into the form suitable for CUDA accelerated computation. In the model checking process, the graph to be searched for accepting cycles is given implicitly. Implicit definition of a graph involves a function to enumerate initial vertices, a function to enumerate edges emanating from a given vertex, and a function to check for accepting status of a given vertex. In order to use our CUDA accelerated accepting cycle detection algorithm, we have to turn the implicit definition of the graph into an explicit one. This process is generally referred to as *state space generation*. In addition to explicit state space construction we also build its CSR representation.

A distinguished property of the MAP algorithm is that it can be altered to work on-the-fly [7]. An on-the-fly algorithm can detect the presence of an accepting cycle before the state space generation procedure completes its task. We were able to adapt our implementation to mimic this behavior as well. In particular, we let the CPU to perform state space generation during which we let the GPU to apply CUDA accelerated MAP algorithm on partially constructed graph. If the part of the graph constructed so far contains an accepting cycle, CUDA accelerated MAP algorithm simply reveals it before the state space generation is complete.

To further accelerate CUDA computation, we employed another technique to decompose the product automaton graph [4], [15]. The idea is to decompose the property automaton into strongly connected components and then project this decomposition to the final graph. Moreover some parts of the product automata graph are known to be without accepting vertices in advance and may be omitted when constructing CSR representation of the graph. This technique significantly reduced the size of the matrix as well as the number of repropagations needed.

## 7. Experimental evaluation

We have implemented the algorithm as a part of the DiVinE-Cluster model checker version 0.8.2 [5]. We compared the performance of the CUDA implementation against the algorithms MAP and OWCTY as provided by the model checker. To order vertices as required by the algorithm, we employed inverse ordering on row numbers since with this ordering the numbers of inner iterations were very small. For the details on how the ordering influences performance of the algorithm, see [8].

To compare the CUDA algorithm with the existing algorithms implemented in the DiVinE Cluster model checker, we used DiVinE native models as listed in Table 1. All the experiments were run on a Linux workstation equipped with two AMD Phenom(tm) II X4 940 Processors @ 3MHz, 8 GB DDR2 @ 1066 MHz RAM and NVIDIA GeForce GTX 280 GPU with 1GB of GPU memory.

Table 2 captures various statistics of our experiments. The difference between *stored* and *generated* states illustrates how much of the state space is made of subgraphs without accepting states. Note that if the graph contains an accepting cycle, the reported numbers refer to numbers of states and transitions generated and stored before the accepting cycle was discovered. *#MAP iterations* reports the number of outer iterations, *#kernel executions* gives the total number of calls to CUDA kernels, and *avg kernel time* gives an average time a single call to a CUDA kernel took.

Table 3 provides details on run-times of individual algorithm parts. As for the CUDA MAP algorithm, the total run-time includes the initialization time (not reported in the table), CSR construction time (*CSR time*), and time spent on CUDA computation (*CUDA time*). Note that the first iteration of CPU MAP is actually

| Models | Model description | Inspected LTL properties |
|---|---|---|
| elevator | the elevator controller | 1: if level 1 is requested, it is served eventually |
| | | 2: if level 1 is requested, it is served as soon as the cab passes the level 1 |
| peterson | Peterson's mutual exclusion algorithm | 1: infinitely many times someone is in the critical section |
| | | 2: if process 0 is not in the critical section then it will eventually reach it |
| leader | leader election algorithm based on filters | eventually leader will be elected |
| anderson | Anderson's queue lock mutual exclusion algorithm | for each process holds that if the process is active infinitely often then it is in the critical section infinitely often |
| bakery | Bakery mutual exclusion algorithm | for each process holds that if the process is active infinitely often and starts wait then it waits until reaches the critical section and eventually reaches it |
| phils | dining philosophers problem | infinitely many times someone eats |

Table 1. The used experimental models.

| Model | *# generated states* | *# stored states* | *# generated transitions* | *# stored transitions* | *accepting cycle* | *# MAP iterations* | *# kernel executions* | *avg. kernel time [ms]* |
|---|---|---|---|---|---|---|---|---|
| elevator 1 | 5 015 528 | 1 722 344 | 63 110 616 | 20 483 544 | N | 14 | 539 | 18 |
| leader | 26 302 351 | 26 302 351 | 84 124 038 | 84 124 038 | N | 2 | 3 | 58 |
| peterson 1 | 18 995 033 | 9 497 514 | 124 897 292 | 41 457 112 | N | 10 | 160 | 40 |
| anderson | 10 728 476 | 6 170 260 | 46 795 735 | 26 328 440 | N | 4 | 223 | 27 |
| elevator 2 | 6 645 826 | 3 354 971 | 76 052 914 | 32 562 797 | Y | 1 | 42 | 22 |
| phils | 6 976 798 | 2 278 932 | 63 492 002 | 7 470 054 | Y | 1 | 1 | 12 |
| peterson 2 | 5 797 524 | 2 933 213 | 38 297 450 | 12 943 640 | Y | 4 | 525 | 11 |
| bakery | 6 986 289 | 4 333 229 | 37 438 316 | 18 145 482 | Y | 1 | 1 | 23 |

Table 2. The statistic of CUDA MAP algorithm.

| | | CUDA MAP | | | CPU MAP | | | | CPU OWCTY | |
|---|---|---|---|---|---|---|---|---|---|---|
| Model | *accepting cycle* | *CSR time* | *CUDA time* | *total time* | *1st iter. time* | *other iter. time* | *total time* | *# iter.* | *reachability time* | *total time* |
| elevator 1 | N | 26 | 7 | 34 | 44 | 56 | 100 | 16 | 24 | 41 |
| leader | N | 87 | 1 | 90 | 97 | 600 | 697 | 17 | 90 | 297 |
| peterson 1 | N | 105 | 6 | 113 | 175 | 270 | 445 | 16 | 110 | 188 |
| anderson | N | 31 | 7 | 39 | 64 | 51 | 115 | 5 | 33 | 113 |
| elevator 2 | Y | 33 | 1 | 35 | 50 | – | 50 | 1 | 41 | 177 |
| phils | Y | 45 | 1 | 47 | 295 | 102 | 397 | 5 | 180 | 576 |
| peterson 2 | Y | 25 | 5 | 31 | 173 | – | 173 | 1 | 114 | 404 |
| bakery | Y | 24 | 1 | 26 | 240 | – | 240 | 1 | 219 | 907 |

Table 3. The run-times in seconds.

| | CUDA MAP | CPU MAP | | CPU OWCTY | |
|---|---|---|---|---|---|
| Models | *total time* | *total time* | *CUDA MAP speedup* | *total time* | *CUDA MAP speedup* |
| non-accepting | 276 | 1357 | 4.92 | 639 | 2.32 |
| accepting | 139 | 860 | 6.19 | 2064 | 14.87 |
| both | 415 | 2173 | 5.24 | 2730 | 6.51 |

Table 4. The overall run-times in seconds, and speedup of the whole model checking procedure.

slower than construction of the CSR representation. This is because the first iteration of the CPU MAP not only generates the state space, but also computes first stable values of *map*. Also note the different number of outer iterations in CUDA MAP (reported in Table 2) and CPU MAP. The difference is a result of employing maximal accepting predecessors in CPU MAP and maximal accepting successors in CUDA MAP. The number of iterations of CUDA MAP is consistently smaller, for which we have no good explanation yet. Algorithms MAP and OWCTY were running on a single core.

Finally, Table 4 gives a comparison of overall runtimes for both valid and invalid model checking instances. We can see that if the whole model checking procedure is considered, the speedup is not that impressive. This is obviously due to the CSR representation preparation. Though, the speedup is still significant.

## 8. Conclusions

We demonstrated successful reformulation of the LTL model checking algorithm MAP in terms of matrix-vector product that allows for significant GPU accelerated model checking process. The main bottleneck of the whole approach is the costly procedure of preparation of data structures that are necessary for efficient acceleration. Though we put significant effort in designing accelerated CSR representation computation, we did not achieve a procedure with consistent speed-up. Therefore, we consider GPU accelerating of the data structures preparation to be the next challenge for model checking community.

We are aware of other representations that could be used for CUDA efficient matrix-vector product, the other representations even exhibit better CUDA performance, however, their preparation is generally more complex, hence not very suitable for our domain.

In the future we would like to accelerate slow CSR representation preparation at least by means of multi-core processing, which we believe may bring similar speed-up as in the case of state space generation [3]. Another problem we are aware of is the limited memory size of a single CUDA device. We intend to overcome this limit by employing multiple CUDA devices for which we already have some initial thoughts.

## References

[1] Christel Baier and Joost P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[2] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Model Checking Software (SPIN 2007)*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[3] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

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

[5] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification (CAV 2006)*, volume 4144/2006 of *LNCS*, pages 278–281. Springer, 2006.

[6] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[7] 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.

[8] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. *ENTCS*, 132(2):3–18, 2006.

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

[10] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[11] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 2.0,. http://www.nvidia.com/object/cuda_develop.html, June 2009.

[12] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[13] Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th annual conference on Design automation (DAC'08)*, pages 2–6. ACM, 2008.

[14] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[15] A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, July 2002.

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

[17] S. Schwoon and J. Esparza. A Note on On-The-Fly Verification Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

[18] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, Januar 1972.

[19] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

# Parallel Algorithms for Finding SCCs in Implicitly Given Graphs[⋆]

Jiří Barnat and Pavel Moravec

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

**Abstract.** We examine existing parallel algorithms for detection of strongly connected components and discuss their applicability to the case when the graph to be decomposed is given implicitly. In particular, we list individual techniques that parallel algorithms for SCC detection are assembled from and show how to assemble a new more efficient algorithm for solving the problem. In the paper we also report on a preliminary experimental study we did to evaluate the new algorithm.

## 1 Introduction

The problem of finding strongly connected components (SCCs), known also as SCC decomposition, is one of the basic graph problems that finds its applications in many research fields even beyond the scope of computer science. An efficient algorithmic solution to this problem is due to Tarjan [20] who showed that given a graph with $n$ vertices and $m$ edges, it is possible to identify and list all strongly connected components of the graph in $O(n+m)$ time and $O(n)$ space. Besides Tarjan's serial algorithm, several parallel algorithms have been designed to solve the problem. Tarjan's algorithm (and its variants) strongly rely on the depth-first search post-ordering of vertices whose computation is known to be $P$-complete [19], and thus, difficult to be parallelized. Therefore, parallel algorithms avoid the depth-first search of the graph and build on different approaches.

A parallel algorithm relying on matrix multiplication was described in [14] and further improved in [10,1]. The algorithm works in $O(log^2 n)$ time in the worst case, however, to achieve the complexity it requires $O(n^{2.376})$ parallel processors. As graphs that we are typically dealing with in practice contain millions of vertices the algorithm is practically unusable and is only interesting from the theoretical point of view. Another parallel algorithm for finding SCCs was given in [12]. Its general idea is to repeatedly pick a vertex of the graph and identify the component the vertex belongs to using two parallel reachability procedures. The algorithm proved to be efficient enough in practice, which resulted in several theoretical improvements of it [17,15]. The worst time complexity of the algorithm is $O(n \cdot (n+m))$, nevertheless, the algorithm exhibits $O(m \cdot log\, n)$ expected time [12].

[⋆] This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338.

In this paper, we discuss known as well as suggest new techniques used for parallel SCC decomposition, and we explore their restrictions if they are applied to implicitly given graphs. Efficient parallel algorithms for SCC decomposition will find their application in distributed formal verification tools such as `DiVinE` [2], `CADP` [13], `DUPPAAL` [4], `LiQuor` [8], etc. Namely, they will allow the tools to verify stochastic systems, compute $\tau$-confluence, or verify systems with fairness constraints or properties given by other than Büchi automata.

The rest of the paper is organized as follows. We recapitulate basic terms and definitions in Section 2, describe known and new techniques used in parallel algorithms for solving the problem in Section 3, and list known parallel algorithms along with their pseudo-codes in Section 4. In Section 5 we report on an experimental study we performed, and in Section 6 we conclude the paper with several remarks and plans for future work.

## 2 Preliminaries

We start by brief summary of basic terms and definitions. Let $V$ be a set of vertices, $E \subseteq V \times V$ a set of directed edges, and $v_0 \in V$ a vertex. We denote by $G = (V, E, v_0)$ a directed graph with initial vertex $v_0$.

Let $G = (V, E, v_0)$ be a directed graph. A sequence of edges $(u_0, u_1), (u_1, u_2), \dots, (u_{n-1}, u_n)$ is called a *path* from vertex $u_0$ to vertex $u_n$. We say that vertex $v$ *is reachable* from vertex $u$ if there is a path from $u$ to $v$ or $u = v$. A *strongly connected component* (SCC) is a subset $C \subseteq V$ such that for any vertices $u, v \in C$ $u$ is reachable from $v$. A strongly connected component $C$ is *maximal* if there is no strongly connected component $C'$ such that $C \subsetneq C'$. A maximal strongly connected component $C$ is *trivial* if $C$ is made of a single vertex $c$ and $(c, c) \notin E$. Henceforward, we speak of maximal strongly connected components as of strongly connected components.

Let $W_G$ be the set of all strongly connected components of graph $G = (V, E, v_0)$. A directed graph of strongly connected components of graph $G$ is defined as $SCC(G) = (W_G, H_G, w_0)$, where $w_0$ is the component that contains the initial vertex $v_0$, and $H_G \subseteq W_G \times W_G$ is the set of edges between members of $W_G$. $(w_1, w_2) \in H_G$ if there are vertices $u_1 \in w_1$ and $u_2 \in w_2$ such that $(u_1, u_2) \in E$. Note that the graph of strongly connected components of any directed graph is acyclic.

A graph could be given in many ways. For purpose of this paper (and according to our needs) we consider graphs that are given implicitly. A graph is given implicitly if it is defined by its initial vertex and a function returning immediate successors of arbitrary vertex. Within the context of implicitly given graphs there are some restrictions the algorithms have to follow. If an algorithm requires any piece of information that cannot be concluded from the implicit definition of the graph, the algorithms have to compute the information first. For example, there is no way to directly identify immediate predecessors of a given vertex from the implicit definition of the graph. If the algorithm needs to enumerate immediate predecessors, then all the predecessors must be computed and stored first.

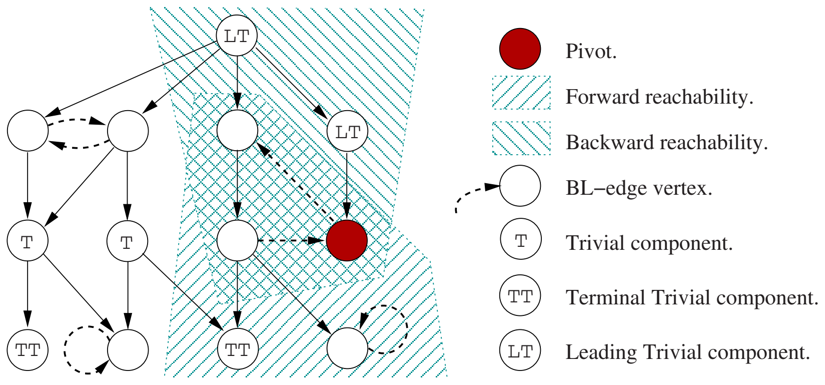


**Fig. 1.** Component detection, identified subgraphs, and trivial components

Similarly, to number vertices of an implicitly given graph means to enumerate all its vertices first. For numbering vertices of implicitly given graphs a parallel procedure was introduced in [13]. Note that vertices of an implicitly given graph are trivially reachable from the initial vertex.

The reason for dealing with implicitly given graphs comes from practice. In many cases, the description of rules according to which the graph could be generated is more space efficient than the enumeration of all vertices and edges. The difference might be quite significant. For example, in the context of model checking [9], the implicit definition of the graph is up to exponentially more succinct compared to the explicit one. This is commonly referred to as the state explosion problem [9].

## 3 List of Techniques

Before describing individual parallel algorithms we give the basic principles and list common techniques that all later given algorithms use. We hope this allow us to describe the algorithms and analyze their behavior in more compact and clear way.

### Basic Principle

All parallel algorithms we present in Section 4 build on the same technique that was originally presented in [12]. The graph to be decomposed is split into two parts. The *decomposed* part of the graph consisting of already identified components, and the *not-yet-decomposed* part of the graph consisting of vertices that have not been classified into strongly connected components yet. The basic step of each algorithm consists of picking a vertex from the *not-yet-decomposed* part of the graph, the so called *pivot*, and identifying the component the selected vertex belongs to. Having a pivot, the strongly connected component the pivot belongs to is determined as the intersection of sets of all predecessors and successors of the given pivot [12]. The structure of all algorithms is then a simple loop in which the basic step is repeated until the *not-yet-decomposed* part becomes empty. The basic step is illustrated on the example graph depicted in Figure 1. Note that the *not-yet-decomposed* part of the graph is further structured as explained below.

## Reachability Relation

Computation of the reachability relation is the core procedure used in all the algorithms. The task of the procedure is to identify all vertices that are reachable from a given vertex. The standard breadth-first or depth-first traversals of the graph can be employed to do so using $O(n)$ space and $O(n + m)$ time.

The reachability procedures are the first place where parallelism appears in the algorithms. The parallelization of a reachability procedure became the standard technique [6,7,21,16]. The so called *partition function* is used to assign every vertex of the graph to a single processor that is responsible for exploration of the vertex. Every processor participating the parallel computation maintains its own set of already explored vertices and its own list of vertices to be explored. If a vertex has been explored previously (it is in the set of explored vertices), then its re-exploration is omitted, otherwise, its immediate successors are generated and distributed into lists of vertices to be explored according to the partition function.

The algorithms we describe in the next section use, in addition to the notion of *forward reachability*, the notion of *backward reachability*. The task of a backward reachability procedure is to identify all vertices that a given vertex can be reached from. The procedure for backward reachability mimics the behavior of the procedure for the forward reachability except it uses immediate predecessors instead of immediate successors during graph traversal. While forward reachability can be performed using only the implicit definition of the graph, the backward reachability, as explained above, requires a list of immediate predecessors to be computed and stored for every vertex first.

## Trivial Strongly Connected Components

Considering the basic algorithmic approach to SCC decomposition, the detection of trivial components is quite inefficient. If the pivot itself is a trivial component, both forward and backward reachability procedures perform useless work. There is rather small improvement in omitting the backward reachability procedure in the case the forward procedure did not hit the pivot, however, the forward procedure still performs $O(n+m)$ work. Therefore, any technique that prevents trivial components from becoming pivots has significant impact on practical performance of the algorithm.

A possible approach for doing so builds on the elimination of leading and terminal trivial components from the *not-yet-decomposed* part of the graph. In particular, every vertex that has zero predecessors must be a trivial component and as such it can be immediately removed (along with all incident edges) from the *not-yet-decomposed* part of the graph. Removing such a vertex may, however, produce new vertices without predecessors that can be removed in the same way. We refer to this recursive elimination technique as to the *One-Way-Catch-Them-Young* elimination (OWCTY) [11]. The technique can be applied in the analogue way also to vertices without successors (Reversed OWCTY). The improved version of the basic parallel algorithm that perform OWCTY elimination procedures before selection of the pivot was described in [15]. We stress that only leading and terminal trivial components may be identified in this way. Trivial components that are neither leading nor terminal may still be chosen as pivots. The graph depicted in Figure 1 contains all three types of trivial components: leading trivial components (`LT`), terminal trivial components (`TT`), and trivial components that are neither leading nor terminal (`T`).

Regarding implicitly given graphs the OWCTY elimination techniques suffer from the difficulty of identifying vertices with zero predecessors or zero successors. Basically, a complete reachability of the *not-yet-decomposed* part of the graph has to be performed to list those vertices. This reachability does not increase the theoretical complexity, however, it may play significant role in the practical performance of the algorithm.

Finally, let us mention that in many cases trivial components of the graph are of a little interest. Therefore, it make sense to save running time by avoiding their explicit enumeration that can be done using a single additional reachability procedure.

**Pivot Selection**

Pivot selection plays a significant role in the complexity of the algorithm. Imagine we always pick a pivot belonging to a component that has no descendant components in the component graph of the *not-yet-decomposed* part. Due to the acyclicity of the component graph such a component always exists. Having such a pivot all vertices belonging to the corresponding component can be identified using only a single forward reachability initiated at the pivot and restricted to the *not-yet-decomposed* part of the graph. Decomposing the graph to SCCs in this manner results in a linear time procedure. Unfortunately, to pick pivots so that the condition above is satisfied means to pick pivots in the depth-first search post-ordering, which is, as stated in the introduction, difficult to be done in parallel. Since the optimal pivot selection is difficult, pivots are typically selected randomly. A random pivot selection leads to $O(m \cdot log\, n)$ expected time as claimed in [12].

In the explicit case, we can presuppose that vertices are numbered. Therefore, picking a random pivot corresponds to the generation of a random number. However, the problem occurs if a pivot has to be selected among vertices of the *not-yet-decomposed* part of the graph. As we are not aware of any $O(1)$ time and $O(1)$ space technique for a single pivot selection, we suggest a technique whose complexity is $O(n)$ space and $O(n)$ time if time and space complexity are summed for all pivot selection procedures called within a single run of the algorithm. The technique is applicable to implicitly given graphs as well. First, each participating processor enqueues newly discovered vertices in a local queue when doing the very first forward reachability of the graph. Then, a new pivot

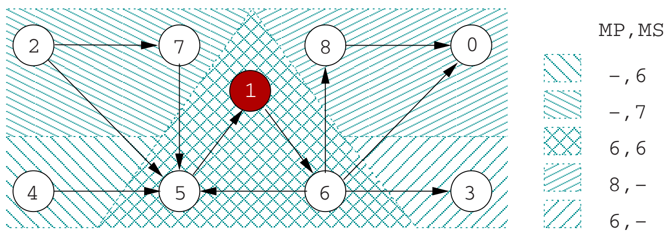


**Fig. 2.** Subgraphs identified with maximal predecessors (MP) and maximal successors (MS) if the propagations of MPs and MSs are initiated at pivot vertex

can be selected among the heads of the local queues. However, if the vertex on the head of a local queue belongs to the *decomposed* part of the graph, it is dequeued and the next head is considered to be a candidate for pivot selection. Moreover, in the case of implicitly given graphs, the procedure organizing vertices into local queues can be combined with the procedure computing the immediate predecessors of vertices producing thus no overhead at all.

As we are typically not interested in trivial components, we suggest a completely new improvement in pivot selection. The idea is to prevent some trivial strongly connected components from being selected as pivots. We achieve this with the definition of the so called *candidate set*, i.e. the set of vertices among which pivots are chosen. Our intention is to terminate the algorithm once all candidate pivots have been selected and the corresponding components identified. If the candidate set contains initially at least one vertex for every non-trivial component of the graph, it must be the case that after the algorithm terminates the remaining *not-yet-decomposed* part of the graph is made of trivial components only. Generally, the smaller the candidate set is, the fewer trivial components are chosen as pivots. What we use for computing the candidate set is the concept of the so called *back-level edge* [3]. It is known that every cycle, and thus every non-trivial strongly connected component, contains at least one back-level edge, which is an edge that leads from a vertex with some distance from the initial vertex of the graph to a vertex with equal or smaller distance from the initial vertex of the graph. Let us call the destination vertex of a back-level edge a *BL-edge* vertex. We suggest the candidate set to be the set of *BL-edge* vertices. Note that *BL-edge* vertices can be computed during the initial reachability procedure using the level-synchronized breadth-first search of the graph [3]. As depicted in the graph in Figure 1, some trivial components can never become pivots considering *BL-edge* vertices as pivot candidates.

## Independent Subgraphs

In every iteration of the outermost loop of the basic algorithm the *not-yet-decomposed* part of the graph is split into several disjoint subgraphs. Let alone the identified component, these are the subgraph induced by vertices out of the component but explored during the forward reachability, subgraph induced by vertices out of the component but explored during the backward reachability,

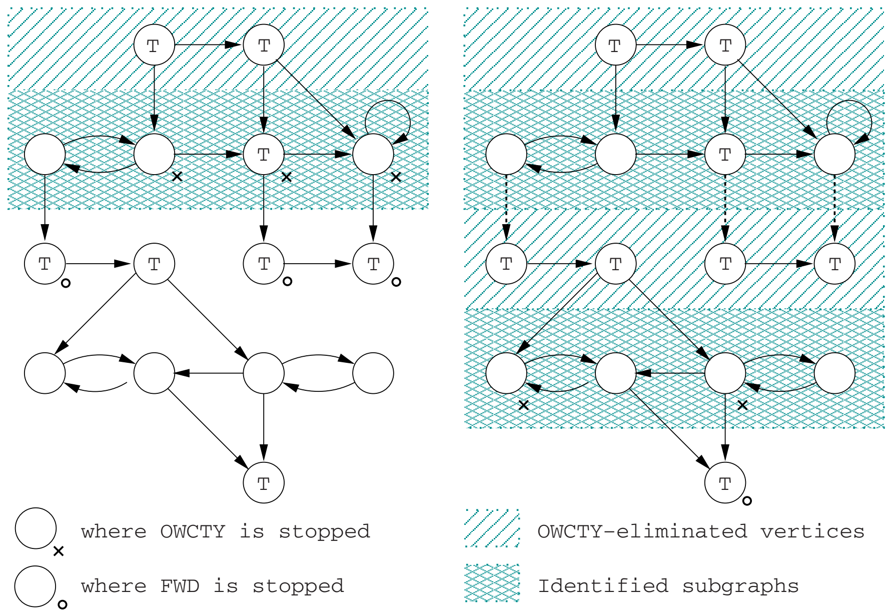


**Fig. 3.** Two steps of OWCTY-BWD-FWD independent subgraph identification

and subgraph induced by vertices that were not explored in that iteration at all. See example in Figure 1. An important observation [12] is that decomposing one of these subgraphs into strongly connected components is completely independent of the decomposition of other subgraphs. Therefore, the subgraphs may be viewed as if they were three independent graphs for the next step of the algorithm, which introduces two major improvements. First, three independent decomposition procedures may be performed in parallel increasing thus the amount of parallelism, second, every independent procedure may be restricted to explore vertices within the subgraph avoiding thus useless exploration of vertices out of the subgraph. Let us call the number of independent subgraphs produced in every iteration of the outermost loop (excluding the identified component) the *degree of parallelism* of the algorithm. Note that the number of the independent subgraphs grows exponentially with the number of iterations. Thus, if $p$ is the number of available processors and $d$ is the degree of parallelism, then after $log_d(p)$ iterations the number of independent subgraphs may exceed the number of available processors.

In the case of implicitly given graphs, vertices of a given subgraph are partitioned among processors according to the partition function. Therefore, all the single decomposition procedures share all processors participating the computation. Unfortunately, this requires to perform as many independent distributed termination detection procedures as there are single decomposition procedures running in parallel, which is quite technically involved and may actually be a reason for preventing individual decomposition procedures from being executed concurrently in practice. Also note that considering independent subgraphs, efficient pivot selection becomes complicated. In particular, we are not aware of any technique that could be used for selection of a random pivot from a subgraph without actually performing the whole subgraph exploration first. Also identifying leading and terminal trivial components in a subgraph results in a reachability procedure performed on the subgraph before the subgraph is decomposed. That is why we did not considered the leading and terminal trivial components elimination in all of the algorithms.

There is a technique that allows to identify more than three subgraphs in a single iteration [17]. Suppose the vertices of the graph are arbitrarily linearly ordered. Then, the maximal preceding vertex and maximal succeeding vertex can be computed for any vertex of the graph using an $O(n \cdot m)$ procedure [5]. If the forward and backward reachability procedures are extended to propagate maximal preceding and succeeding vertices, respectively, new subgraphs can be identified according to the maximal predecessors and successors associated to vertices in the subgraph. All vertices of the strongly connected component that the selected pivot belongs to, must have the same maximal predecessor and successor. Due to the pivot selection the maximal predecessors are computed only in the subgraph induced by vertices reachable from the pivot (forward reachability) while the maximal successors are computed for vertices that can reach the pivot (backward reachability). A possible result after a single iteration on a subgraph is depicted in Figure 2. In the original approach described in [17], the maximal predecessors and successors were computed over the complete graph. Regarding the number of identified components none of the approaches is better.

In the following we suggest a completely new technique to identify independent subgraphs in $O(n+m)$ time. The technique employs OWCTY elimination technique succeeded with backward and forward reachability procedures. A graph and two steps of the new technique performed on the graph are depicted in Figure 3. The OWCTY elimination procedure, if initiated from the vertex with zero predecessors, eliminates all leading trivial components and visits some vertices of all components immediately reachable from the eliminated trivial ones. Visited but not eliminated vertices are shown as vertices with a little cross. A backward reachability performed from vertices with the little cross identifies one independent subgraph. Note that this subgraph contains exactly all strongly connected components immediately reachable from the eliminated trivial components. Having the subgraph a forward reachability procedure restricted to the subgraph is performed from the vertices with little cross. This procedure stops on vertices outside the subgraph but immediately reachable from the subgraph (vertices with the little circle). Among these vertices there might be some that have predecessors only in the previously identified subgraph. These vertices must be trivial components, and therefore, they can be used as vertices to start the next OWCTY elimination procedure from. Figure 3 shows two successive steps of the subgraph identifying procedure OWCTY-BWD-FWD. We stress that the procedure may detect many independent subgraphs while performing only $O(n+m)$ work.

## 4 Algorithms

Having described all the techniques, we can now present individual algorithms. All pseudo-codes listed below describe the core parts of the algorithms. We neither list the initial reachability procedure that must be performed in order to compute the predecessor function in the implicit case, nor we list many technical details related to implementation, parallelization, distribution, etc.

### F-B

The F-B algorithm [12] is the basic algorithm that all other presented algorithms build on. In the following pseudo-code, we describe a single procedure that is initially called for the complete set of vertices of the graph to be decomposed and then called recursively for identified independent subgraphs. A pivot is selected using procedure PIVOT and the set of vertices reachable in forward and backward manner are computed using parallel reachability procedures FWD and BWD. Both reachability procedures have two parameters. Besides the vertex or vertices to start from, each reachability procedure is also given a set of vertices that its exploration is limited to. This ensures that given a subgraph, the procedure will explore only immediate successors or predecessor of vertices within the subgraph. The sets of vertices as computed by forward and backward reachability procedures are referred to as $F$ and $B$, respectively. Having computed both sets $F$ and $B$, a new component is identified as the intersection of $F$ and $B$, and recursive calls for three new subgraphs are made. Note that if it is necessary, the procedure is able to select pivots among given set of candidates.

```
1  proc F-B($V, candidates$)
2    if ($V \neq \emptyset$)
3      then $p$ := PIVOT($V \cap candidates$)
4           $F$ := FWD($p, V$)
5           $B$ := BWD($p, V$)
6           $SCCs := SCCs \cup \{F \cap B\}$
7           in parallel do
8             F-B($F \setminus B, candidates$)
9             F-B($B \setminus F, candidates$)
10            F-B($V \setminus (F \cup B), candidates$)
11          od
12   fi
13 end
```

In our experimental study we also considered a slightly modified version of the basic algorithm. In particular, we implemented a version in which the backward reachability procedure was restricted to the vertices discovered by the preceding forward reachability procedure, i.e. line 5 in the pseudo-code is changed to

$$B := \mathrm{BWD}(p, F).$$

This modification decrease the degree of parallelism, but produce a procedure where exploration of some vertices is omitted compared to the original algorithm. One could tend to see this technique as an improvement, however, our experiments proved that neither of the versions was significantly better then the other. Our explanation for the lack of improvement in some cases is that the subgraphs identified as $(V \setminus (F \cup B))$ become actually larger causing thus more work to be done in subsequent recursive calls to the procedure.

## MP-MS

Algorithm MP-MS [17] extends the previous algorithm with the maximal predecessors and maximal successors concept. Compared to algorithm F-B, the improvement is in the subgraph detection, see Section 3. In order to compute the maximal predecessors and successors, parallel procedures FWD and BWD have to be replaced with new parallel procedures FWD-MAXPRED and BWD-MAXSUCC, respectively. Besides computing the same reachability relation as procedures FWD and BWD, the new procedures also identify subgraphs according to the maximality of predecessors or successor and return lists of those vertices whose order is used to refer to a subgraph. These *SuccList* and *PredList* are then used to perform parallel recursive calls on identified subgraphs. See the pseudo-code below. Recall that the time complexity of both new procedures is $O(n \cdot m)$, which is worse than if simple reachability procedures are used. However, the bad complexity is paid off with the degree of parallelism being much higher compared to the degree of parallelism of algorithm F-B. Finally, let us mention that also algorithm MP-MS is capable of selecting pivots among given set of candidates.

```
proc MP-MS($V$, $candidates$)
  if ($V \neq \emptyset$)
    then $p :=$ PIVOT($V \cap candidates$)
         $F, PredList :=$ FWD-MAXPRED($p, V$)
         $B, SuccList :=$ BWD-MAXSUCC($p, V$)
         $SCCs := SCCs \cup \{F \cap B\}$
         in parallel do
           MP-MS($V \setminus (F \cup B), candidates$)
           MP-MS($F[k, -], candidates$) foreach $k \in PredList$
           MP-MS($B[-, k], candidates$) foreach $k \in SuccList$
         od
  fi
end
```

## O-B-F

Algorithm O-B-F is completely a new algorithm we suggest in this paper. The core idea of the algorithm is to partition the component graph to the so called *O-B-F levels* and then call any algorithm (F-B in our case) to decompose individual levels of the component graph into strongly connected components. Recall that the component graph can be partitioned to those levels in linear time using the new technique described in Section 3.

The procedure O-B-F performs the detection of levels in the level by level manner. It is started with the complete set of vertices as the graph to be decomposed, and with the initial vertex as the vertex to start the exploration from. In every single call of the procedure one O-B-F level is detected. The set of remaining vertices, denoted with $V$, is appropriately shrunk, candidates for initial vertices of $V$ (the so called *Seeds*) are computed, and two procedures are initiated in parallel. First, a procedure to decompose the subgraph identified with the O-B-F level, second, procedure O-B-F to identify other levels in the remaining set $V$. Recursive calls to procedure O-B-F terminates when all the levels are recognized and the set of remaining vertices is empty.

Every single O-B-F level is detected using standard procedures. First of all, leading trivial components of the remaining graph are eliminated using procedure OWCTY. The procedure computes the set of leading trivial components (*Eliminated*) and the set of vertices on which the elimination process stopped (*Reached*). Eliminated vertices are removed from set $V$ of remaining vertices and the standard backward reachability procedure is performed from vertices in *Reached* and restricted to vertices in $V$. As the backward procedure is restricted to $V$, it computes exactly vertices belonging to the top most level in the component graph of $V$. These vertices (denoted with $B$) are removed from set $V$ of remaining vertices and the decomposition of the level is initiated as an independent parallel procedure. Note that the set of potential pivots can be restricted to vertices in *Reached* because every strongly connected component belonging to the level must contain at least one vertex from *Reached*. A forward reachability (FWD-SEEDS) is also performed on vertices in $B$ in order to identify vertices immediately below the current level, which are exactly vertices that become *Seeds* for the next call to procedure OWCTY in the next recursive call of the procedure O-B-F. Note that vertices in *Seeds* that belong to non-trivial strongly connected components are moved directly to set *Reached* within procedure OWCTY.

```
proc O-B-F($V$, $Seeds$)
  if ($V \neq \emptyset$)
    then $Eliminated, Reached =$ OWCTY($Seeds, V$)
         $V := V \setminus Eliminated$
         $B :=$ BWD($Reached, V$)
         $V := V \setminus \{B\}$
         $F, Seeds :=$ FWD-SEEDS($Reached, B$)
         in parallel do
           F-B($B, B$)
           O-B-F($V, Seeds$)
         od
  fi
end
```

## 5 Experimental Evaluation

We have implemented and experimentally evaluated quite a few algorithms described in this paper. The algorithms were implemented using the DiVinE Library [2] as the library providing support for parallel and distributed generation of implicitly given graphs. The common library used gives approximately the same level of enhancement of all implementations, thus, the experimental comparison is quite fair. All experiments were conducted on a network of ten Intel Pentium 4 2.6 GHz workstations each having 1 GB of RAM and 100Mbps switched Ethernet connection.

The graphs we use to evaluate our implementations come from DiVinE Library distribution. They are listed in Table 1 along with their important characteristics, namely, the number of vertices (**Vertices**), number of edges (**Edges**), numbers of trivial and non-trivial strongly connected components (**T. SCCs**, **N. SCCs**), and the time needed for sequential decomposition into strongly connected components using Tarjan's algorithm (**Tarjan**). Value *n.a.* means that the sequential decomposition algorithm exceeded available amount of RAM. For the purpose of the distributed experiments, all the graphs were distributed using the default hash-based partition function implemented in DiVinE Library.

We implemented and experimentally evaluated six different algorithms. Algorithms **F-B**, **MP-MS**, and **O-B-F** directly correspond to algorithms presented in Section 4. Algorithm **F-RB** is the modified version of algorithm **F-B**, i.e. the version where the backward reachability procedure is restricted to vertices explored during the preceding forward reachability procedure. If the name of the algorithm is extended with suffix **(B)**, then the algorithm was initiated considering *BL-edge* vertices as pivot candidates. We have not implemented the modification of algorithms **F-B** and **MP-MS** that includes elimination of leading and terminating trivial components on the given subgraph before the subgraph is decomposed [15,18]. The reason is that we are not aware of any technique that would identify vertices with zero predecessors or zero successors in the given subgraph without actually exploring the subgraph first, which makes the approach inefficient in the case of implicitly given graphs.

All our implementations explicitly avoid concurrent performance of the decomposition procedures on independent subgraphs. In particular, if an independent decomposition procedure is about to be initiated, its assignment is stored and its initiation postponed. There are several reasons for this. First, the number of processors we have at our hand is very limited. Therefore, parallel procedures would very soon produce a non-trivial overhead caused by switching the context of CPUs depreciating thus the measured values. Second, as already mentioned in Section 3, appropriate termination detection becomes technically involved if independent parallel procedures share CPUs. Moreover, pivot selection within the given subgraph would generally introduce additional reachability procedure performed on every discovered independent subgraph if the subgraphs should be decomposed in parallel. And third, as the algorithms perform parallel reachability procedures most of the time, we have not observed idling of individual workstations. Therefore, we believe that the parallelism of the decomposition

**Table 1.** Summary of graphs

| Name | Vertices | Edges | T. SCCs | N. SCCs | Tarjan |
|---|---|---|---|---|---|
| DrivingPhilsK3 | 6307240 | 12950475 | 16 | 1 | 4:51 |
| DrivingPhilsK3_4 | 10301529 | 24055321 | 3170354 | 2680 | 10:27 |
| Elevator12_2 | 8591334 | 89419176 | 2004966 | 2 | 13:21 |
| Lifts6 | 16364845 | 50088312 | 7231789 | 8052 | n.a. |
| LookUpProc10_3 | 16562363 | 33464135 | 1603283 | 2 | n.a. |
| MutBak4 | 9384762 | 31630895 | 1881088 | 15 | 30:07 |
| MutMcs4 | 1241948 | 4456310 | 9718 | 39 | 33 |
| Phils14_1 | 9565935 | 124357142 | 531442 | 28 | n.a. |
| Pet6err | 1060048 | 6656522 | 208436 | 25075 | 4:29 |
| Rether9_2 | 7663993 | 9624242 | 81831 | 5 | 16:38 |
| Train8_2 | 11740214 | 37389502 | 5273750 | 50858 | 3:10:44 |

**Table 2.** Runtimes (hours:minutes:seconds)

| Graph | F-B | F-B (B) | F-RB (B) | MP-MS | MP-MS (B) | O-B-F |
|---|---|---|---|---|---|---|
| DrivingPhilsK3 | 2:13 | 1:58 | 2:08 | 17:20 | 22:43 | 1:57 |
| DrivingPhilsK3_4 | n.a. | 3:41:37 | n.a. | n.a. | n.a. | 4:30:36 |
| Elevator12_2 | n.a. | n.a. | n.a. | n.a. | n.a. | 9:06 |
| Lifts6 | n.a. | 5:15:46 | n.a. | n.a. | n.a. | 5:47:44 |
| LookUpProc10_3 | n.a. | n.a. | n.a. | n.a. | n.a. | 16:36 |
| MutBak4 | n.a. | 2:31:31 | 1:42:31 | n.a. | n.a. | 1:29:09 |
| MutMcs4 | 7:32 | 37 | 23 | 26:21 | 34:32 | 23 |
| Phils14_1 | 2:42:03 | 18:36 | 18:30 | n.a. | n.a. | 21:31 |
| Pet6err | n.a. | n.a. | n.a. | n.a. | n.a. | 4:53:47 |
| Rether9_2 | 1:13:01 | 27:57 | 8:23 | 4:13:29 | 2:14:05 | 17:11 |
| Train8_2 | n.a. | 2:09:54 | 1:52:21 | n.a. | n.a. | n.a. |

procedures would bring nothing but increased complexity of the implementations. Actual runtimes needed by all the algorithms to decompose the graphs are reported in Table 2. Value *n.a.* denotes now that the runtime of the algorithm exceeded 10 hours time limit.

We find the experimental results very interesting. First, we were slightly surprised with the practical inefficiency of the algorithm based on maximal predecessors and maximal successors. Its performance is far beyond performance of other algorithms proving that the decomposition into many subgraphs is not worth unless it is done in $O(n + m)$ time. Second, quite interesting result is that the restriction of the set of vertices that can be selected for pivots play significant role in practice. Note that in the case of algorithm **F-B**, the *BL-edge* vertices yielded roughly speed up of to ten. In the case of algorithm **MP-MS** they did not generally help at all, for which we blame the procedures with $O(n \cdot m)$ time complexity whose bad performance cut off any improvements made in pivot selection. Third, algorithm **O-B-F** proved to have big potential as it was the fastest algorithm in many cases, and sometimes even the only algorithm that was able to perform the decomposition within the given time limit. Finally, let us mention that according to our preliminary experiments, there were cases where the parallel algorithms if executed on ten workstations, outperformed even the optimal Tarjan's algorithm.

## 6 Conclusion and Future Work

In this paper we tried to list and evaluate all known techniques used in parallel algorithms for decomposition of implicitly given graphs into strongly connected components, and compare the parallel algorithms that exploit them. We also introduced two completely new techniques that the parallel algorithms can employ. In particular, we suggested how *BL-edge* vertices can be profited from if they are used as pivot candidates, and how the graph can be decomposed into subgraphs preserving SCCs using linear time and parallel technique OWCTY-BWD-FWD. Both newly suggested techniques have shown their superior strength in our experimental study.

We would especially like to emphasize that the newly suggested procedure shows not only practical usefulness, but also a theoretically interesting behavior. In particular, it may be proved that graphs whose components exhibit a chain-like structure, can be decomposed in parallel in the optimal linear time. Generally, using the technique we are able to give a parallel algorithm for solving the SCC decomposition problem working in $O(h \cdot (n + m))$ time, where $h$ is the maximal number of strongly connected components on an acyclic path in a single *O-B-F* level.

Although, the preliminary results are encouraging, we are well aware of the immaturity of our experimental evaluation. We intend to perform thorough experimental study on larger set of inputs including algorithms with elimination of leading and terminal trivial components in the future. We also intend to improve implementations of the algorithms, in particularly, we would like to come up with a reasonable pivot selection procedure that would allow us to implement and experimentally evaluate virtually concurrent decomposition of the independent subgraphs. Finally, we intend to incorporate the best algorithms in the distributed verification environment DiVinE, so that the tool is capable of distributed and parallel verification of stochastic systems as well as verification of properties given by other than Büchi automata.

Let us also mention that we have tried to come up with some algorithms that avoid backward reachability procedure being thus perfectly suitable for the decomposition of implicitly given graphs. However, all our attempts resulted in algorithms whose practical performance was quite poor and discouraging.

## References

1. Nancy Amato. Improved processor bounds for parallel algorithms for weighted directed graphs. *Inf. Process. Lett.*, 45(3):147–152, 1993.
2. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine – a tool for distributed verification. To appear in proceedings of CAV 2006.

3. 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.
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, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.
6. S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. De Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935 of *LNCS*, pages 181–200. Springer-Verlag, 1995.
7. G. Ciardo, J. Gluckman, and D.M. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 1997.
8. Frank Ciesinski and Christel Baier. LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems, 2006.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
10. Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989.
11. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.
12. Lisa K. Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. *Lecture Notes in Computer Science*, 1800:505–511, 2000.
13. H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.
14. Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988.
15. William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, 2005.
16. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
17. S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.
18. S.M. Orzan and J.C. van de Pol. Detecting strongly connected components in large distributed state spaces. Technical Report SEN-E0501, CWI, 2005.
19. John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.
20. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, pages 146–160, 1972.
21. U.Stern and D. L. Dill. Parallelizing the mur$\varphi$ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.

**Journal of Logic and Computation Advance Access published February 17, 2009**

# Distributed Algorithms for SCC Decomposition

JIŘÍ BARNAT and JAKUB CHALOUPKA, *Department of Computer Science, Faculty of Informatics, Masaryk University Brno, Czech Republic.*
*E-mail: barnat@fi.muni.cz; xchalou1@fi.muni.cz*

JACO VAN DE POL, *Formal Methods and Tools, Department of EEMCS, University of Twente, The Netherlands.*
*E-mail: j.c.vandepol@ewi.utwente.nl*

## Abstract

We study existing parallel algorithms for the decomposition of a partitioned graph into its strongly connected components (SCCs). In particular, we identify several individual procedures that the algorithms are assembled from and show how to assemble a new and more efficient algorithm, called Recursive OBF (OBFR), to solve the decomposition problem. We also report on a thorough experimental study to evaluate the new algorithm. It shows that it is possible to perform SCC decomposition in parallel efficiently and that OBFR, if properly implemented, is the best choice in most cases.

*Keywords*: parallel algorithms, strongly connected components

## 1 Introduction

The problem of finding strongly connected components (SCCs), known also as SCC decomposition, is one of the basic graph problems that finds its applications in many research fields, even beyond the scope of computer science. An efficient algorithmic solution to this problem is due to Tarjan [25], who showed that, given a graph with $n$ vertices and $m$ edges, it is possible to identify and list all SCCs of the graph in $O(n+m)$ time and $O(n)$ space.

Among many applications, the algorithm may be used also for the analysis of computer systems. In particular, algorithms for SCC decomposition find their application in distributed formal verification tools such as `CADP` [18], `DiVinE` [2], `DUPPAAL` [5], `LiQuor` [12], $\mu$`CRL` [6], etc. Namely, they allow the tools to verify quantitative properties of probabilistic systems, compute $\tau$-confluence [8], form a pre-processing step for branching bisimulation reduction, or verify systems with fairness constraints or properties given by extensions of Büchi automata.

Unfortunately, graphs modelling complex computer systems tend to be very large, which makes it hard to handle them on a single machine. One way to tackle this problem is to distribute the graph across a cluster of workstations and employ a distributed algorithm to decompose the partitioned graph. However, Tarjan's algorithm (and all other linear algorithms for SCC decomposition, e.g. Kosaraju's algorithm also known as Double DFS [15]) strongly rely on the depth-first search post-ordering of vertices, whose computation is known to be $P$-complete [23], and thus, difficult to be computed in parallel. Therefore, different approaches have been used to design parallel algorithms for solving the problem.

doi:10.1093/logcom/exp003

A parallel algorithm based on matrix multiplication was described in [19] and further improved in [1, 14]. The algorithm works in $O(log^2 n)$ time in the worst case. However, to achieve this low time complexity it requires $O(n^{2.376})$ parallel processors. As typical graphs that we are interested in contain millions of vertices the algorithm is practically unusable and is only interesting from a theoretical point of view. Another parallel algorithm for finding SCCs was given in [17]. It exploits the fact that it is possible to efficiently compute the set of vertices reachable from a certain vertex or set of vertices in parallel. The general idea of the algorithm is to repeatedly pick a vertex of the graph and identify the component to which it belongs, by using a forward and a backward parallel reachability procedure. The algorithm proved to be efficient enough in practice, which resulted in several theoretical improvements of it [20, 22]. The worst-time complexity of the algorithm is $O(n\cdot(n+m))$. Nevertheless, the algorithm exhibits $O(m\cdot logn)$ expected time [17]. Another algorithm was introduced in [22]. That algorithm is more involved, but still, its basic building block is a simple parallel value iteration technique.

In this article, which can be viewed as a full version of [3, 4], we summarize a number of known procedures used for distributed SCC decomposition. Moreover, we present a new algorithm based on re-arranging these procedures, and extensively compare its implementation with existing algorithms. The rest of the article is organized as follows. We recapitulate basic terms and definitions in Section 2, describe known techniques and algorithms for solving SCC decomposition in Section 3. The new algorithm based on recursive application of OBF [3, 4] is described in Section 4. Compared with [3], we added full proofs for the correctness and the complexity claims. Results of experiments are in Section 5. In particular, we compare our new algorithm with the algorithms from [17, 22], and we measure the effect of decomposing sub-graphs one by one, or in parallel. Contributions of the article are summarized and future work is outlined in Section 6.

## 2 Preliminaries

### 2.1 Directed graphs

A (directed) graph G is a pair (V, E), where V is a set of vertices, and $E\subseteq V\times V$ is a set of (directed) edges. If $(u,v)\in E$, then $v$ is called (immediate) successor of $u$ and $u$ is called (immediate) predecessor of $v$. The *indegree* of a vertex $v$ is the number of immediate predecessors of $v$. $G^T=(V,E^T)$, the *transposed graph* of $G=(V,E)$, is the graph $G$ with all edges reversed, i.e. $E^T=\{(u,v)\mid(v,u)\in E\}$.

Let $G=(V,E)$ be a directed graph. Let $E^*$ be a transitive and reflexive closure of $E$ and $s,t\in V$ two vertices. We say that vertex $t$ *is reachable* from vertex $s$ if $(s,t)\in E^*$. If $s_k$ is reachable from $s_0$, then there is a sequence of vertices $s_0,\dots,s_k$, s.t. $(s_i,s_{i+1})\in E$ for all $0\leq i<k$. We call this sequence a *path*. A path is *simple* if it contains no duplicated vertices. The *length* of the path is $k$, i.e. the number of edges. A graph is *rooted* if there is an initial vertex $s_0\in V$ such that all vertices in $V$ are reachable from $s_0$. Given a graph $G$, we use $n$, $m$ and $l$, to denote the number of vertices and edges, and the length of the longest simple path between any two vertices in $G$, respectively.

A set of vertices $C\subseteq V$ is *strongly connected*, if for any vertices $u,v\in C$, we have that $v$ is reachable from $u$. A strongly connected component (SCC) is a *maximal* strongly connected $C\subseteq V$, i.e. such that no $C'$ with $C\subsetneq C'\subseteq V$ is strongly connected. A maximal SCC $C$ is *trivial* if $C$ is made of a single vertex $c$ and $(c,c)\notin E$, and is *non-trivial* otherwise. Henceforward, an SCC is also referred to simply as a component.

Let $W_G$ be the set of all SSCs of graph $G=(V,E)$. The *quotient graph* of graph $G$ is a directed graph $SCC(G)=(W_G,H_G)$, where $H_G=\{(w_1,w_2)\mid(\exists u_1,u_2\in V)(u_1\in w_1\wedge u_2\in w_2\wedge(u_1,u_2)\in E)\}$, i.e. there is an edge between SCCs if and only if there is an edge between some members of the SCCs in the original graph. Note that the quotient graph of any directed graph is acyclic. Given a graph $G$, we denote by $N$, $M$ and $L$, the number of vertices and edges, and the length of the longest (simple) path in the quotient graph of $G$, respectively. An SCC is *leading* if it has no predecessors in the quotient graph. A set $S \subseteq V$ is *SCC-closed* if each SCC in the graph is either completely inside the set or completely outside the set; such $S$ is also referred to as an *independent sub-graph*.

For $v \in W \subseteq V$, the *forward closure* of $v$ in $W$ is the set of reachable states from $v$ in the graph $(V, E_W)$, where $E_W = \{(x,y) \mid (x,y) \in E \wedge x,y \in W\}$. If $W$ is not specified, the whole graph is meant. The forward closure of $S \subseteq W$ in $W$ is the union of forward closures of all vertices from $S$ in $W$. Finally, the *backward closure* of $v$ (or $S$) in $W$ is the forward closure of $v$ (or $S$) in $W$ in the graph $G^T$.

## 2.2 Graph representation

A directed graph can be given in many ways. We restrict ourselves to explicit vertex representations, excluding symbolic representations, e.g. based on binary decision diagrams.

Beside the standard representations by adjacency lists or an adjacency matrix we also mention graphs that are given *implicitly* (do not confuse with symbolic representation, this is still an explicit vertex representation). A rooted graph is given implicitly if it is defined by its initial vertex and a function returning immediate successors of an arbitrary vertex. Within the context of implicitly given graphs there are some restrictions that algorithms have to follow. If an algorithm requires any piece of information that cannot be concluded from the implicit definition of the graph, it has to compute the information first. For example, there is no way to directly identify immediate predecessors of a given vertex from the implicit definition of the graph. If the algorithm needs to enumerate immediate predecessors, then the predecessors must be stored, while enumerating the whole graph first. Similarly, in order to number the vertices of an implicitly given graph, one must enumerate all its vertices first. For numbering the vertices of implicitly given graphs a parallel procedure was introduced in [18]. Note that all vertices of an implicitly given graph are reachable from the initial vertex by definition.

The reason for dealing with implicitly given graphs comes from practice. In many cases, the description of rules according to which the graph can be generated is more space efficient than the enumeration of all vertices and edges. The difference might be quite significant. For example, in the context of model checking [13], the implicit definition of the graph is up to exponentially more succinct compared with the explicit one. This is commonly referred to as the state explosion problem [13]. However, it turns out that, in the situation where the graph has to be traversed more than once, which is the case for all parallel SCC decomposition algorithms, it is advantageous to first generate the whole graph and store it in an explicit form. All subsequent computations are then performed using the explicit representation. We save the time for repeated generation of successors and since the graphs we are interested in are mainly sparse, the needed memory is proportional to the number of vertices only.

## 3 Known algorithms

Before describing individual parallel algorithms, we describe the basic techniques that the later algorithms will use. This allows us to describe the algorithms and analyse their behaviour in a more compact and clearer way.

All parallel algorithms presented in this article build on the same basic principle. The graph to be decomposed is divided into independent (SCC-closed) sub-graphs. These are further divided into smaller independent sub-graphs until they become SCCs. All the algorithms take advantage of the fact that computation on separate independent sub-graphs can be done in parallel.

## *3.1 Reachability relation*

Computation of the reachability relation is the core procedure used in all the algorithms. The task of the procedure is to identify all vertices that are reachable from a given vertex, i.e. to compute its forward closure. The standard breadth-first or depth-first traversals of the graph can be employed to do so using $O(n)$ space and $O(n+m)$ time.

The reachability procedure is the first place where parallelism comes into play in the algorithms. The parallelization of a reachability procedure has by now become a standard technique [10, 11, 21, 24]. A so called *partition function* is used to assign vertices to processors. Each processor is responsible for the exploration of the vertices assigned to it by the partition function. Each processor maintains its own set of already visited vertices and its own list of vertices to be explored. If a vertex has been visited previously (it is in the set of visited vertices), then its re-exploration is omitted. Otherwise, its immediate successors are generated and distributed into lists of vertices, to be explored according to the partition function.

The algorithms described in the next section use the notion of *backward reachability*, in addition to the notion of *forward reachability*. The task of a backward reachability procedure is to identify all vertices that a given vertex can be reached from. The procedure for backward reachability mimics the behaviour of the procedure for the forward reachability except it uses immediate predecessors instead of immediate successors during graph traversal.

Note that in many cases, the forward and backward reachability procedure are restricted to a particular independent sub-graph of the original graph. This can be achieved by an additional marking of that sub-graph, or simply by deleting edges that leave that sub-graph.

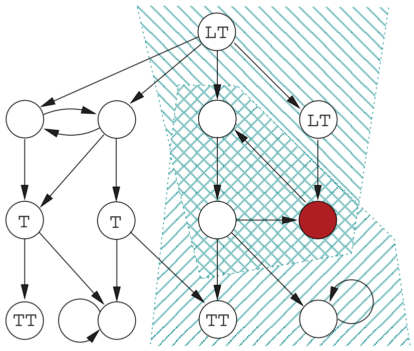
## *3.2 Pivot selection*

In several algorithms, there is a point at which a certain vertex (called pivot) must be selected from the current independent sub-graph to start the decomposition of that sub-graph. Pivot selection plays a significant role in the complexity of the algorithms. Imagine, we always pick a pivot belonging to a component that has no descendant components in the component graph of the sub-graph being decomposed. Due to the acyclicity of the component graph such a component always exists. Having such a pivot, all vertices belonging to the corresponding component can be identified using only a single forward reachability initiated at the pivot. Decomposing the graph to SCCs in this manner results in a linear-time procedure. Unfortunately, to pick pivots so that the condition above is satisfied means to pick pivots in the depth-first search post-ordering, which is, as stated in the Section 1, difficult to be done in parallel. Since the optimal pivot selection is difficult, pivots are typically selected randomly.

## *3.3 Trivial SCCs*

This sub-section presents an efficient technique for the elimination of leading and terminal trivial (LT and TT, respectively) components from any independent sub-graph. Use of this technique can significantly speed up all the SCC decomposition algorithms, since they are not that efficient on detecting trivial components.

Every vertex that has zero predecessors must be a trivial component and as such it can be immediately removed (along with all incident edges) from the graph. Removing such a vertex may, however, produce new vertices without predecessors that can be removed in the same way. We refer to this recursive elimination technique as *One-Way-Catch-Them-Young* (OWCTY) elimination [16]. The technique can be applied in an analogous way to vertices without successors (Reversed OWCTY)



| | |
|---|---|
| ● | Pivot. |
| ▨ | Forward reachability. |
| ▨ | Backward reachability. |
| T | Trivial component. |
| TT | Terminal Trivial component. |
| LT | Leading Trivial component. |

FIGURE 1. Component detection, identified sub-graphs and trivial components

as well. An improved version of the basic parallel algorithm that performs OWCTY elimination before selection of the pivot was described in [20]. We stress that only LT and TT components may be identified in this way. Trivial components in between non-trivial SCCs will not be identified. These components, however, may become leading or terminal when the graph is further divided. The graph depicted in Figure 1 contains all three types of trivial components: LT, TT and trivial components that are neither leading nor terminal (T).

Having described the basic techniques, we can now present individual algorithms. All pseudocodes listed below describe the core parts of the algorithms. We neither list the initial reachability procedure that must be performed in order to compute the explicit representation from the implicit one, nor the many technical details related to implementation, parallelization, distribution, etc.

## *3.4 FB*

The FB algorithm [17] is the basic algorithm, outlined in Section 1. We illustrate the basic principle of this algorithm. Figure 1 shows the basic step of the algorithm. First, a vertex (called *pivot*) is selected at random from an independent sub-graph (the whole graph in this situation) that is not known to be a single SCC yet. Second, the forward and the backward closure of the pivot are computed; these are depicted by shaded regions. This procedure divides the graph into four independent sub-graphs. The vertices that are both in the forward and the backward closure form the SCC of pivot and need not be further processed. The other three sub-graphs are: vertices in the forward closure but not in the backward closure, vertices in the backward closure but not in the forward closure and vertices that are neither in the forward nor in the backward closure. These three sub-graphs have to be further decomposed. They can be decomposed independently and hence in parallel. Recursive application of the basic step is used to do it.

The pseudocode of the algorithm is in Figure 2. A pivot is selected using procedure PIVOT and its forward and backward closures are computed using parallel reachability procedures FWD and BWD. Both reachability procedures have two parameters. Besides the vertex or vertices to start from, each reachability procedure is also given a set of vertices that its exploration is limited to. This ensures that given a sub-graph, the procedure will explore only immediate successors or predecessor of vertices within the sub-graph. The sets of vertices as computed by forward and backward reachability procedures are referred to as $F$ and $B$, respectively. Having computed both sets $F$ and $B$, a new component is identified as the intersection of $F$ and $B$, and recursive calls for three new subgraphs are made. As stated in Section 1, the time complexity of the algorithm is $O(n\cdot(n+m))$.

```
proc FB($V$)
  if ($V \neq \emptyset$)
    then $p$ := PIVOT($V$)
    $F$ := FWD($p$, $V$)
    $B$ := BWD($p$, $V$)
    $F \cap B$ is SCC
    in parallel do
      FB($F \setminus B$)
      FB($B \setminus F$)
      FB($V \setminus (F \cup B)$)
    od
  fi
end
```

FIGURE 2. FB algorithm

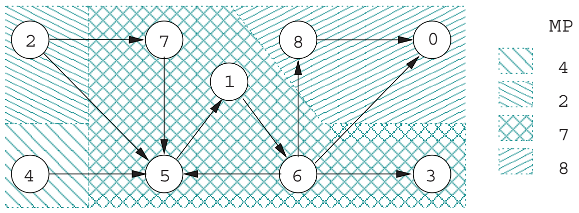


FIGURE 3. Sub-graphs identified with maximal predecessors

## 3.5 *Colouring/heads-off*

The colouring algorithm was introduced in [22]. It uses a totally ordered set of colours. Initially, each vertex has its own colour. The colours are repeatedly propagated to successors with a smaller colour, until all edges are non-decreasing. A forward reachability procedure augmented to propagate maximal visited colours can be used for this task. Note that a vertex can be re-coloured several times, which results in time complexity of $O(n \cdot m)$ [9]. The final colour of a vertex is the colour of its maximal predecessor, i.e. predecessor with maximal colour. Here a predecessor does not necessarily mean an immediate predecessor (as in the rest of this article), but here it means any vertex in the backward closure. After colouring, all vertices in a single SCC have the same colour. This is because all vertices in a single SCC share the same set of predecessors. So all edges between vertices of different colours can be removed. This technique is able to divide the graph into more than four parts, as opposed to the technique presented in Sub-section 3.4. Unfortunately, we do not know how to do this in linear time. A graph division obtained after colouring is depicted in Figure 3.

In the second step, one takes as roots those vertices that kept their initial colour. The SCC of each root consists of those vertices that are backward reachable (within the same colour) from it. These SCCs are removed (heads-off) and the algorithm proceeds with the remaining sub-graph and with the original colour assignment.

The pseudocode of the algorithm is in Figure 4. Computation of maximal predecessors is done by the procedure FWD-MAXPRED, which returns the list of roots as *PredList*. It also computes for each $k \in PredList$ the set $V_k$ of vertices with maximal predecessor $k$. The SCCs of the roots are identified by the standard procedure BWD, which performs backward reachability. The removal of these SCCs

```
proc CH($V$)
  if ($V \neq \emptyset$)
    then $PredList, (V_k)_{k \in PredList}$ := FWD-MAXPRED($V$)
    foreach $k \in PredList$ do
      in parallel do
        $B_k$ := BWD($k, V_k$)
        $B_k$ is SCC
        CH($V_k \setminus B_k$)
      od
    od
  fi
end
```

FIGURE 4. Colouring/heads-off (CH) algorithm

```
proc OBF($V, v$)
  $Seeds := \{v\}$
  [$v$ is initial vertex]
  while $V \neq \emptyset$ do
    $Eliminated, Reached$ := OWCTY($Seeds, V$)
    $V := V \setminus Eliminated$
    [All elements of $Eliminated$ are trivial SCCs]
    $B$ := BWD($Reached, V$)
    in parallel do
      FB($B$)
    od
    $Seeds$ := FWD-SEEDS($B, V$)
    $V := V \setminus B$
  od
end
```

FIGURE 5. OBF algorithm

on line 8 was referred to as heads-off in the previous paragraph. Edges are not removed there. Instead, separate recursive calls of the main procedure restricted to the appropriate sub-graphs are used.

The time complexity of the algorithm is $O((L+1)\cdot n\cdot m))$, where $O(n\cdot m)$ comes from the complexity of the FWD-MAXPRED procedure. The total complexity follows from the fact that every time a recursive call is invoked, it is on a graph with strictly shorter longest path in the quotient graph.

## 3.6 OBF

This algorithm is based on a recent technique OWCTY-BWD-FWD (OBF) [3, 4] which gave name to the whole algorithm. It identifies a number of independent sub-graphs (called *OBF slices*) in $O(n+m)$ time. The slices are then decomposed using the FB algorithm. This algorithm assumes the input graph to be rooted, i.e. we have an initial vertex from which all other vertices are reachable.

The OBF technique repeatedly employs OWCTY elimination, succeeded with backward and forward reachability. Each iteration identifies one OBF slice. The pseudocode of the algorithm is in Figure 5. A graph and two steps of the technique performed on the graph are depicted in Figure 6.

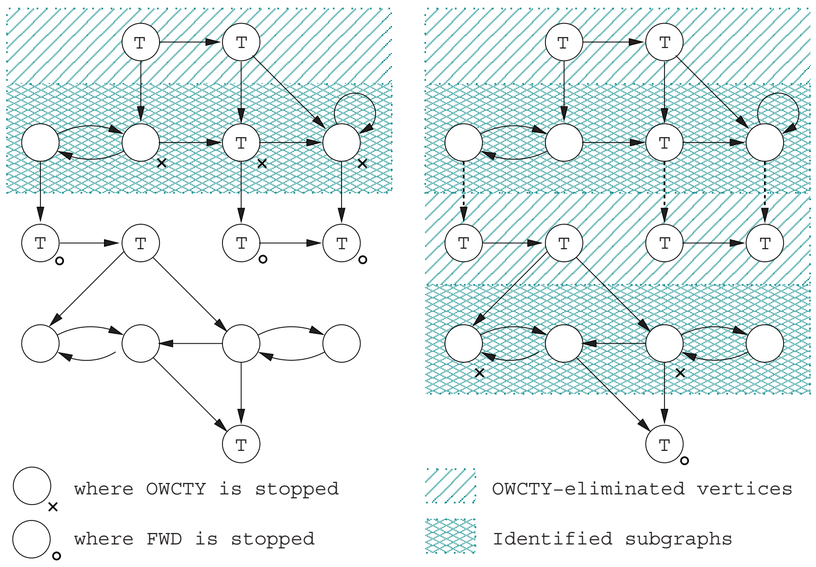


FIGURE 6. Two steps of BF independent sub-graph identification

We simultaneously describe the figure and the pseudocode. We start with the initial vertex (the vertex with no predecessors in the figure, the vertex $v$ in the pseudocode). The OWCTY elimination procedure (line 5 in pseudocode) eliminates all LT components (the set *Eliminated* in the pseudocode) and visits some vertices of all components immediately reachable from the eliminated trivial ones. Visited but not eliminated vertices are shown as vertices with a little cross (the set *Reached*). A backward reachability (BWD()) performed from vertices with the little cross identifies the first OBF slice (the set $B$). Note that the slice contains exactly all SCC immediately reachable from the eliminated trivial components. The decomposition of the slice is initiated as an independent parallel procedure (line 10). Then a forward reachability procedure that stops on immediate successors of vertices in the slice is executed (FWD-SEEDS()). These successors (vertices with the little circle in the figure, *Seeds* on line 12 in the pseudocode) are used to start the next iteration of OBF. The time complexity of the algorithm is $O(n\cdot(n+m))$, the same as for the FB algorithm.

## 4 Recursive OBF

As shown in [4], OBF performs better than FB in a number of experiments. Note that in OBF the graph is split into slices in linear time. On each slice, algorithm FB is applied. But, as OBF is better than FB, we now propose to *recursively apply* OBF to the slices.

However, the slice may not be rooted, so we must:

- repeatedly pick a vertex from the slice and compute its forward closure within the slice; we call this a ‘rooted chunk’. Subsequently run OBF on each rooted chunk within the slice;
- add a termination criterion in case the whole slice is one SCC.

Adding a termination criterion is easy. No special work has to be done. We simply count the vertices visited during the first backward search in the first rooted chunk (The ‘B’ part of OBF). If the slice consists of exactly one SCC there will be only one rooted chunk in it; O will not eliminate any vertex,

```
1  proc OBFR-P($V$)
2    while ($V \neq \emptyset$) do
3      $v$ := PIVOT($V$)
4      $Range := \mathrm{FWD}(v, V)$
5      $Seeds := \{v\}$
6      $V := V \setminus Range$
7      $OriginalRange := Range$
8      while $Range \neq \emptyset$ do
9        [Invariant: The forward closure of $Seeds$ in $Range = Range$]
10       $Eliminated, Reached := \mathrm{OWCTY}(Seeds, Range)$
11       $Range := Range \setminus Eliminated$
12       [All elements of $Eliminated$ are trivial SCCs]
13       $B := \mathrm{BWD}(Reached, Range)$
14       if ($B = OriginalRange$) then
15         $B$ is SCC
16       else
17         in parallel do
18           OBFR-P($B$)
19         od
20         $Seeds := \mathrm{FWD\text{-}SEEDS}(B, Range)$
21       fi
22       $Range := Range \setminus B$
23     od
24   od
25 end
```

FIGURE 7. OBFR

and so B will be started from the root and explores the whole slice. Conversely, if B starting from the root of the first chunk explores the whole slice, the slice is one SCC, for it is both the forward and the backward closure of the root. We now describe Recursive OBF (OBFR) in more detail.

The pseudocode of OBFR is in Figure 7. The suffix '-P' in the name of the procedure means that it runs in parallel on independent subgraphs. The term OBFR without any suffixes is used to refer to Recursive OBF as such, without specifying the degree of parallelism (see Sub-section 4.1).

We start with the whole graph. Vertices in recognized SCCs are removed from the 'working' set $V$ until we end up with an empty set at which point all SCCs have been identified.

Initially, we assume that we do not have a vertex from which all other vertices are reachable (initial vertex). To start OBF we need such a vertex, so we pick one vertex (line 3) and compute its forward closure *Range* in $V$ using procedure FWD() (line 4). OBF is then applied on *Range*. Vertices from $V \setminus Range$ will be processed in the next iterations of the main while-loop (lines 2–24).

Before OBF is started on *Range*, *Range* is saved into *OriginalRange*, this will enable us to determine if a slice found by OBF is an SCC. Of course, in the actual implementation we only store the size of *OriginalRange*. On line 9, there is an invariant '(The forward closure of *Seeds* in *Range*) = *Range*'. In the first iteration of the while-loop on lines 8–23 the invariant holds trivially, because *Seeds* contains just one vertex and *Range* was computed as a forward closure of that vertex. Procedure OWCTY() eliminates LT components by repeatedly removing indegree 0 vertices reachable from *Seeds*. Eliminated vertices are returned as the set *Eliminated*, and subsequently removed from *Range*. Vertices at which OWCTY() stops (they have positive indegree) are returned as the set *Reached*. The forward closure of *Reached* in *Range* equals *Range*, since any path that leads from *Seeds* to a non-eliminated vertex has to contain some vertex from *Reached*. All elements from *Eliminated* are trivial SCCs. Now a backward search is started from vertices in *Reached*. This search is implemented by procedure BWD(). Backward closure of *Reached* in *Range* is returned as the set *B*. This is the first SCC-closed slice found by OBF. If the set *B* equals the set *OriginalRange*, it means that all vertices in the SCC-closed set *OriginalRange* are reachable from the same single vertex (note that $B = OriginalRange$ is only possible in the first iteration of the while-loop 8–23) and so *B* is indeed an SCC. Consequently, $Range \setminus B$ is the empty set and the while-loop finishes.

If $B \neq OriginalRange$ we run OBFR-P() on *B* recursively. Moreover, note that the nested procedure can be run in parallel, which increases parallelism. *Seeds* for the next iteration of the while-loop 8–23 are computed by the procedure FWD-SEEDS, which simply returns all vertices from *Range* that are immediate successors of vertices in *B* but not in *B*. Since all paths that reach vertices in $Range \setminus B$ from *B* must contain some vertex from *Seeds*, after we subtract *B* from *Range*, the invariant of line 9 is satisfied. When $Range = \emptyset$, the while-loop 8–23 finishes and we handle the remaining vertices in *V*.

We now formally prove the correctness of the algorithm. The key point is the invariant on line 9. It ensures that the whole graph is eventually processed. As argued earlier, it trivially holds in the first iteration of the while-loop on lines 8–23. Thus, it remains to show that, if the invariant holds in iteration *i*, then it holds also in iteration $i+1$. Together with the fact than *Range* gets smaller in every iteration, it implies that the whole rooted chunk computed on line 4 is processed on lines 8–23. Another important point is that the set *B* computed on line 13 is an independent (SCC-closed) sub-graph. This implies partial correctness. Since line 18 is executed only if *B* is smaller than *OriginalRange*, finite depth of recursion and thus termination of the algorithm is ensured. All the statements in this paragraph are proved below.

We sometimes use a set of vertices to refer to the graph induced by that set. To prove the invariant, we need to strengthen it a bit. In addition to the fact that the forward closure of *Seeds* in *Range* is equal to *Range*, we argue that *Range* is an independent sub-graph of *OriginalRange*. Since initially $Range = OriginalRange$, the strengthened invariant holds in the first iteration of the while-loop.

The following lemmata analyse one iteration of the while-loop on lines 8–23. In the whole iteration *Range* is used to refer to the set *Range* on line 9, i.e. at the very beginning of the iteration. The same goes for *Seeds*. The set *Range* computed on line 11 is referred to as $Range'$. The set *Range* computed on line 22 is referred to as $Range''$. The set *Seeds* computed on line 20 is referred to as $Seeds'$.

Lemma 1
Vertices eliminated by OWCTY() (the set *Eliminated* on line 10) are trivial SCCs of *OriginalRange*.

Proof. Let us suppose, for the sake of contradiction, that OWCTY() eliminates a vertex $v$ such that there is a vertex $v'$ such that there is a path in *OriginalRange* from $v$ to $v'$ and vice versa. *Range* is an independent sub-graph of *OriginalRange*. It follows, that *Range* contains a cycle $c = (v_0, v_1, \ldots, v_k)$ with $v_0 = v_k = v$. At the moment when $v$ was eliminated it must have had indegree 0, which means that vertex $v_{k-1}$ must have been eliminated earlier, since there is an edge from $v_{k-1}$ to $v$. By repeating this argument, we get that all vertices $v_{k-2}, v_{k-3}, \ldots, v_0$ were eliminated before $v$ and since $v_0 = v$, it means that $v$ was eliminated before $v$. An obvious contradiction. ■

Lemma 2
Let *Reached* be the set of vertices at which OWCTY() stops (cf. line 10; these are the non-eliminated vertices from *Seeds* and the non-eliminated successors of the eliminated vertices). Then the forward closure of *Reached* in $Range'$ is equal to $Range'$.

Proof. Since the forward closure of *Seeds* in *Range* is equal to *Range*, for each $v \in Range'$ there is $w \in Seeds$ such that there is a path $p = (v_0, v_1, \ldots, v_k)$, where $v_0 = w$, $v_k = v$ and $k \geq 0$. Since OWCTY()

eliminates only indegree 0 vertices, there is $j \geq 0$ such that vertices $v_0, \ldots, v_{j-1}$ were eliminated and vertices $v_j, \ldots, v_k$ were not, and vertex $v_j$ is in the set *Reached*. It follows that $v$ is reachable from $v_j$ in *Range*$'$. Therefore, the forward closure of *Reached* in *Range*$'$ is *Range*$'$. ■

LEMMA 3
The set $B$ computed on line 13 (The backward closure of *Reached* in *Range*$'$) is an independent sub-graph of *Range*$'$. (No SCC has vertices both in $B$ and *Range*$' \setminus B$).

PROOF. It is sufficient to show that there is no edge from *Range*$' \setminus B$ to $B$. However, that is obvious for the existence of such edge $(w, v)$ would imply that $w \in B$, which is impossible since, according to the assumption, $w \in$ *Range*$' \setminus B$. ■

LEMMA 4
Let *Seeds*$'$ be the successors of the vertices in $B$ which are in *Range*$'' =$ *Range*$' \setminus B$. Then the forward closure of *Seeds*$'$ in *Range*$''$ is *Range*$''$.

PROOF. Since *Reached* $\subseteq B$, the forward closure of $B$ in *Range*$'$ is *Range*$'$ by Lemma 2. Therefore, for each vertex $v \in$ *Range*$''$ there is $w \in B$ such that there is a path $p = (v_0, v_1, \ldots, v_k)$, where $v_0 = w$, $v_k = v$ and $k \geq 1$. Let $j$ be the greatest index with the property that $v_j \in B$, then $v_{j+1} \in$ *Seeds*$'$ and the path $p' = (v_{j+1}, \ldots, v_k)$ is a path in *Range*$''$. Thus $v$ is reachable from $v_{j+1}$ in *Range*$''$. It follows that the forward closure of *Seeds*$'$ in *Range*$''$ is equal to *Range*$''$. Together with the fact that *Range*$''$ is *Range* without some independent sub-graphs (Lemmas 1 and 3) it implies that if *Range*$'' \neq \emptyset$, then the strengthened invariant is satisfied in the next iteration. ■

So far, we proved the strengthened invariant of line 9 by analysing one iteration of the while-loop on lines 8–23. It follows that the whole set *OriginalRange* computed on line 4 is eventually processed and divided into independent subgraphs by the while-loop. To prove the correctness of the algorithm, we still need to show that it correctly identifies an SCC when it sees it and that it never creates a sub-graph that is not independent, part of which was already shown.

LEMMA 5
If the set *OriginalRange* on line 7 is an independent sub-graph of the whole input graph then *OriginalRange* is an SCC of the whole input graph if and only if, for an arbitrary vertex $v \in$ *OriginalRange*, the forward closure of $v$ in *OriginalRange* is equal to *OriginalRange*, OWCTY($\{v\}$, *OriginalRange*) does not eliminate any vertex, and the backward closure of $v$ in *OriginalRange* is equal to *OriginalRange*.

PROOF. Forward implication. If *OriginalRange* is an SCC, then for each pair of vertices $z, w \in$ *OriginalRange* there is a path from $z$ to $w$ in *OriginalRange*. The statements for the forward and the backward closures follow directly. There is a vertex $w \in$ *OriginalRange* such that there is a path from $w$ to $v$ in *OriginalRange*, so *indegree*$(v) > 0$, and so OWCTY() started from $v$ cannot eliminate any vertex.

Backward implication. For each pair of vertices $z, w \in$ *OriginalRange* there is a path from $z$ to $w$ in *OriginalRange*, which follows from the assumption about the forward and the backward closures. (There is a path from $z$ to $v$ and a path from $v$ to $w$). Since *OriginalRange* is an independent sub-graph of the whole input graph (Lemma 3), *OriginalRange* is an SCC of the whole input graph. ■

LEMMA 6
Let $G = (V, E)$ be an arbitrary graph. For arbitrary vertex $v \in V$, the forward closure of $v$ in $V$, denoted by $A$, is an independent sub-graph of $G$.

PROOF. Similar to the proof of Lemma 3. (There is no edge from $A$ to $V \setminus A$.) ■

THEOREM 1
The algorithm in Figure 7 correctly identifies all SCCs in the input graph.

PROOF. In the while-loop on lines 2–24 the graph is correctly divided into independent sub-graphs by repeated application of lines 3 and 4 (Lemma 6). The sub-graphs that are SCCs are correctly identified by Lemma 5. The sub-graphs that are not SCCs are divided into smaller independent sub-graphs by Lemmas 1–4. To these smaller sub-graphs, the procedure is applied recursively. The only case when the recursive application is not executed is the case when $B = \mathit{OriginalRange}$, which can happen only in the first iteration of the while-loop on lines 8–23. This is exactly the case when *OriginalRange* is one SCC, again by Lemma 5. The rest follows from the fact that the relation 'being an independent subgraph of' is transitive. ■

LEMMA 7
The overall time complexity of Recursive OBF is $\mathcal{O}((r+1)\cdot(m+n))$, where $r$ is the maximal depth of recursion ($r=0$ if no recursive calls are executed).

PROOF. Two distinct OBFR procedures on the same depth of recursion operate on disjoint parts of the graph, so at most $\mathcal{O}(m+n)$ work is done for each recursion depth. Thus the overall complexity is $\mathcal{O}((r+1)\cdot(m+n))$. ■

THEOREM 2
The depth of recursion of Recursive OBF is at most $L$ (the length of the longest path in the quotient graph of the whole graph).

PROOF. The proof proceeds by induction on $L$.
Induction basis. If $L=0$, then the whole graph is one SCC. This is detected on the recursion level zero, so the maximal depth of recursion is 0.
Induction step. It is sufficient to show that application of the procedure in Figure 7 (not counting recursive calls) to a graph with $L=k>0$ divides it into sub-graphs with $L$ at most $k-1$. There are two possible cases.

CASE 1
The SCC of the vertex $v$ selected on line 3 is not the first vertex of any of the longest paths in the quotient graph. Then, obviously, the forward closure of $v$ is an independent sub-graph the quotient graph of which does not contain paths longer than $k-1$. The same goes for all independent sub-graphs into which it might be further divided in the while-loop on lines 8–23.

CASE 2
The SCC of the vertex $v$ selected on line 3 is the first vertex of one of the longest paths in the quotient graph. Then at least one of the longest paths is in the quotient graph of the forward closure of $v$. The important point is that all longest paths in the quotient graph of the forward closure must have the SCC of $v$ as their first vertex. (The path not containing the SCC of $v$ can be extended, because the SCC of $v$ is a leading SCC). If the SCC of $v$ is trivial, it is eliminated by OWCTY. If it is non-trivial, it is equal to the first OBF slice. In both cases, the SCC of $v$ is removed in the first iteration of the while-loop on lines 8–23. Which leaves us with a graph with $L$ less than $k$. The rest follows easily. ■

COROLLARY 1
The overall time complexity of Recursive OBF is $\mathcal{O}((L+1)\cdot(m+n))$.

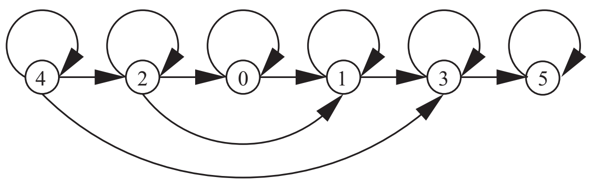


FIGURE 8. Example for lower bound of OBFR

The upper bound cannot be tightened as shown by the following example. Define $\mathcal{G}_k = (V_k, E_k)$ as follows. Let

$$\begin{aligned}
V'_0 &= \{0\}\\
V'_{i+1} &= V'_i \cup \{2i+1, 2i+2\}\\
E'_0 &= \{(0,0)\}\\
E'_{i+1} &= E'_i \cup \{\, (2i+1, 2i+1), (2i+2, 2i+2),\\
&\qquad (\max(2i-1,0), 2i+1), (2i+2, 2i), (2i+2, 2i+1)\}
\end{aligned}$$

for $i \in \{0,\dots,k\}$. Now

$$\begin{aligned}
V_k &= V'_k \cup \{2k+1\}\\
E_k &= E'_k \cup \{(2k+1, 2k+1), (\max(2k-1,0), 2k+1)\}
\end{aligned}$$

Figure 8 shows $\mathcal{G}_2$. Note that $\mathcal{G}_k$ has $2k+2$ vertices and $5k+3$ edges. One possible behaviour of OBFR on $\mathcal{G}_k$ is as follows. Suppose OBFR picks the vertex $2k$ first. All vertices of $\mathcal{G}_k$ are reachable from $2k$ so the first rooted chunk is the whole graph. OBF is then run on this rooted chunk. No vertex is eliminated by OWCTY(), for $2k$ has a predecessor (itself). The first OBF slice is then $\{2k\}$ which is identified as an SCC by subsequent recursive call to OBFR. The first OBF then continues on successors of $\{2k\}$, these are $2k-2$ and $2k-1$. Again, OWCTY() does not eliminate anything. Then a backward reachability is started from $\{2k-2, 2k-1\}$ and explores the whole remaining graph except for the vertex $2k+1$. So, the second OBF slice is equal to the graph $\mathcal{G}_{k-1}$ and OBFR is called recursively to process it.

We have shown that maximal recursion depth of *OBFR* on $\mathcal{G}_k$ is $k+1$. At recursion depth $i$, a graph with at least $2(k-i)+2$ vertices and at least $5(k-i)+3$ edges is explored at least once. So by Corollary 1, the overall time complexity of OBFR on $\mathcal{G}_k$ is $\Omega(n \cdot (n+m))$.

## 4.1 *Increasing the degree of parallelism*

In [4], it was noticed that OBF has a better worst-case running time than CH, mainly due to possible re-colouring. Still, our initial experiments (cf. Figure 11) showed that CH performs better on graphs with many small SCCs. We attribute this to the higher degree of parallelism in CH, which outweighs the extra costs due to re-colouring in this case.

There is room to increase parallelism in OBFR-P() too. The pseudocode of this 'more parallel' version is in Figure 9. It exploits the fact that, after we pick a vertex in $V$ and identify its forward closure *Range* in $V$, we can run OBF on *Range* in parallel and without waiting for its completion we can pick another vertex from $V$ and start computing its closure.

So we essentially have three versions of OBFR varying in the 'degree of parallelism'. This is illustrated in Figure 10. Each diagram starts with a bold vertical axis, where the downward direction represents the progression of time. The numbered columns represent independent parallel procedures.

```
proc OBFR-MP($V$)
  while ($V \neq \emptyset$) do
    Pick a vertex $v \in V$
    $Range := FWD(v, V)$
    $Seeds := \{v\}$
    $V := V \setminus Range$
    in parallel do
      OBFR-MPX($Seeds, Range$)
    od
  od
end
proc OBFR-MPX($Seeds, Range$)
  $OriginalRange := Range$
  while $Range \neq \emptyset$ do
    $Eliminated, Reached, Range :=$ OWCTY($Seeds, Range$)
    All elements of Eliminated are trivial SCCs
    $B :=$ BWD($Reached, Range$)
    if ($B = OriginalRange$) then
      $B$ is SCC
    else
      in parallel do
        OBFR-MP($B$)
      od
      $Seeds :=$ FWD-SEEDS($B, Range$)
    fi
    $Range := Range \setminus B$
  od
end
```

FIGURE 9. OBFR with increased parallelism

An arrow from column *i* to column *j* indicates that procedure *i* starts procedure *j*. For simplicity, the figure does not show recursive calls of OBF.

Assume we have a graph whose vertices are partitioned into the following disjoint sets according to how OBFR works on the graph: $V = B_{11} \cup B_{12} \cup B_{13} \cup B_{21} \cup B_{31} \cup B_{32}$. $B_{1(1-3)} = B_{11} \cup B_{12} \cup B_{13}$ is the closure (*Range*) of the first picked vertex (first rooted chunk) and the individual sets are the slices identified by OBF in the closure. Similarly $B_{2(1)} = B_{21}$ is the closure of the second picked vertex (second rooted chunk) and $B_{3(1-2)} = B_{31} \cup B_{32}$ is the closure of the third picked vertex (third rooted chunk). For simplicity, we assume there are no trivial components eliminated by OWCTY.

The leftmost diagram in Figure 10 illustrates the operation of the basic OBFR when no parallel procedures are executed. SCCs are processed one by one (delete lines 17 and 19 from Figure 7).

The middle diagram in Figure 10 illustrates the operation of OBFR in Figure 7. Each time a new slice is identified by OBF, a new parallel procedure is started to process the slice. The algorithm first picks a vertex, identifies the set $B_{1(1-3)}$, then the slices $B_{11}$, $B_{12}$ and $B_{13}$. Only then it can pick another vertex from the unexplored part of the graph, identify $B_{2(1)}$, ...

The rightmost diagram in Figure 10 illustrates the operation of the 'more parallel' OBFR in Figure 9. It does slicing of $B_{1(1-3)}$, $B_{2(1)}$ and $B_{3(1-2)}$ in separate parallel procedures. This allows it to get to $B_{2(1)}$ and $B_{3(1-2)}$ much faster.

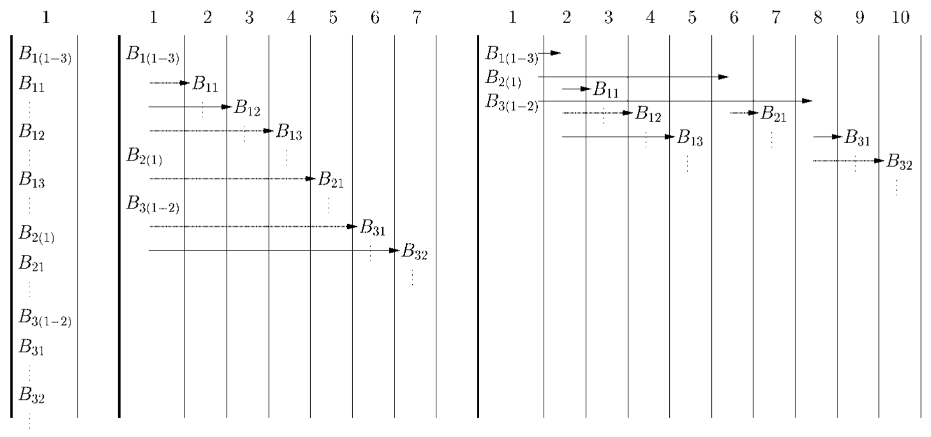


FIGURE 10. Three versions of OBFR different in degree of parallelism

## 5 Experimental evaluation

The experiments were carried out on a cluster of eight workstations interconnected with 1 Gbps Ethernet. Each workstation was equipped with AMD Athlon$^{TM}$ 64 3500+ Processor and 1 GB RAM. We used the LAM/MPI library for message passing. Our implementation is a distributed memory one. The graph is partitioned into a number (in our case 8) of disjoint parts. Each workstation owns one part. Each workstation runs the same code and communicates with other workstations via the message passing library only. The computation at each workstation proceeds sequentially (the execution of independent parallel procedures is serialized) meaning that no additional threads are executed. This is achieved by maintaining an appropriate piece of information about each procedure in an 'array of procedures' and iterating over its elements repeatedly to let each procedure perform some work. Note that a single procedure runs in parallel over different partitions of the graph.

We observed that OBFR suffers from the amount of synchronization points among individual procedures. However, the amount of synchronization points may be significantly reduced if independent procedures are started as soon as all data they depend on are ready. Starting independent procedures can be viewed as an implementation detail, however, it has proven to have significant impact on the performance. The three different versions presented in the previous section are recapitulated in the following.

| | |
|---|---|
| OBFR-S | No procedures are executed in parallel. When OBF identifies a slice it waits for the complete computation on the slice to finish before continuing. |
| OBFR-P | OBF identifies the slices, and starts a parallel procedure on each slice as soon as the slice is identified. |
| OBFR-MP | Does the same as the previous one, but additionally, within a slice, it starts a parallel procedure as soon as a new forward chunk (forward closure of a picked vertex in a possibly not-rooted slice) within a slice is found. |

Our experiments show that indeed the total running time of the algorithm decreases by adding more parallelism, despite the extra overhead (e.g. running various termination detection procedures in parallel), and despite the fact that a single reachability computation is already parallel.

We compare OBFR with three other algorithms. Namely FB [17], OBF + FB [4] and CH (colouring [22]). Like OBFR, FB and OBF + FB can be implemented with different degrees of parallelism. For the comparisons we implemented only the most parallel versions of these algorithms, which give the best results. These implementations are denoted by FB-P and OBF-FB-P. CH processes SCCs inherently in parallel; we reused the code from [22] and all experiments are carried out in the same software/hardware environment.

## 5.1 *Measurements*

For the evaluation, we used synthetic graphs with a regular structure and fixed size SCCs. The aim was to find out how the algorithms work as the SCC size changes. We used two types of graphs. The first type of graph, called L$m$L$m$T$n$ was of the form $Loop(m)$ || $Loop(m)$ || $Tree(n)$, where $Loop(m)$ is a cycle with $m$ states, $Tree(n)$ is the binary tree of depth $n$ and || denotes the Cartesian product of graphs. This graph has $2^{n+1}-1$ components of size $(m+1)^2$. Its quotient graph is a binary tree.

The second type of graph, called Li$m$Lo$n$, uses $Line(m)$, being a sequence of $m$ states. It is of the form $Line(m)$ || $Line(m)$ || $Loop(n)$ || $Loop(n)$ and consequently has $m^2$ components of size $n^2$. The quotient graph of the second type is a square mesh with edges oriented right and down. In the second type, there are many paths of the same length to the same vertex.

We also experimented with graphs that arise as state spaces in real model checking applications. The names of these graphs are prefixed with ‘cwi’, ‘vasy’ and ‘swp’. The former two are taken from the VLTS Benchmark Suite [7][1] The swp-graph, called swp_d$m$w$n$q$p$, models the behaviour of a sliding window protocol with $m$ distinct data elements, window size $2n$, and queue size $p$. The complete list is in Tables 1 and 2.

The size of the graphs is relatively small and in principle they could be decomposed on a single machine, but they are large enough for experiments with distributed algorithms to provide insight.

The results for synthetic graphs are in Table 3. The results for real graphs are in Table 4. All run-times are in seconds, ‘n/a’ means that the run-time exceeded 36 000 s (10 h). Graphs of dependency of run-time on SCC size are in Figure 11 and 12. We measured this dependency for synthetic graphs only. Figure 11 does not contain results for all graphs of type 1 since numbers of vertices of some of these graphs differ too much. Only graphs with $\sim 3000000$ vertices were chosen. The graphs of type 2 have all approximately $\sim 4000000$ vertices, so Figure 12 contains results for all of them.

## 5.2 *Evaluation*

There is one important issue concerning space complexity. To implement a reachability analysis in linear time, we need a way to determine whether a vertex has been already visited or not in constant time. This is usually accomplished by allocating an array of booleans with $n$ elements, one for each vertex. Algorithms that perform many reachabilities in parallel must have such an array for each of them. Our implementations that fall into this category are FB-P, OBF-FB-P, OBFR-P, OBFR-MP. There is no problem with reachabilities in the same depth of recursion. Since they operate on disjoint parts of the graph, one array of size $n$ is enough. But for procedures in different depths we need separate arrays. And so the space complexity is $O(m+n\cdot(\textit{maximum depth of recursion}))$.

Although the maximum depth of recursion can be as high as $n$, in our experiments the algorithm we are mainly interested in, OBFR, reached maximum depth of 15. This makes us believe that

[1]Note that we consider the graph of *all* transitions, while [22] considered only (*invisible*) $\tau$-transitions.

TABLE 1. Synthetic graphs used in experiments

| State space | Number of SCCs | Size of one SCC | States | Transitions |
|---|---:|---:|---:|---:|
| L10L10T10 | 2047 | 121 | 247 687 | 742 940 |
| L100L100T4 | 31 | 10 201 | 316 231 | 938 492 |
| L15L15T10 | 2047 | 256 | 524 032 | 1 571 840 |
| L4L4T16 | 131 071 | 25 | 3 276 775 | 9 830 300 |
| L20L20T12 | 8191 | 441 | 3 612 231 | 10 836 252 |
| L80L80T8 | 511 | 6561 | 3 352 671 | 10 051 452 |
| L350L350T4 | 31 | 123 201 | 3 819 231 | 11 334 492 |
| L1750L1750T0 | 1 | 3066 001 | 3 066 001 | 6 132 002 |
| L1750L1750T1 | 3 | 3066 001 | 9 198 003 | 24 528 008 |
| Li200Lo10 | 40 000 | 100 | 4 000 000 | 15 960 000 |
| Li125Lo16 | 15 625 | 256 | 4 000 000 | 15 936 000 |
| Li100Lo20 | 10 000 | 400 | 4 000 000 | 15 920 000 |
| Li80Lo25 | 6400 | 625 | 4 000 000 | 15 900 000 |
| Li67Lo30 | 4489 | 900 | 4 040 100 | 16 039 800 |
| Li50Lo40 | 2500 | 1600 | 4 000 000 | 15 840 000 |
| Li40Lo50 | 1600 | 2500 | 4 000 000 | 15 800 000 |
| Li30Lo67 | 900 | 4489 | 4 040 100 | 15 891 060 |
| Li25Lo80 | 625 | 6400 | 4 000 000 | 15 680 000 |
| Li20Lo100 | 400 | 10 000 | 4 000 000 | 15 600 000 |
| Li16Lo125 | 256 | 15 625 | 4 000 000 | 15 500 000 |
| Li10Lo200 | 100 | 40 000 | 4 000 000 | 15 200 000 |

TABLE 2. Real graphs used in experiments

| State space | Number of SCCs | Maximum SCC size | States | Transitions |
|---|---:|---:|---:|---:|
| cwi_2165_8723 | 47 926 | 423 505 | 2 165 446 | 8 723 465 |
| cwi_2416_17605 | 2 150 392 | 6 | 2 416 632 | 17 605 592 |
| cwi_7838_59101 | 1 | 7 838 608 | 7 838 608 | 59 101 007 |
| vasy_11026_24660 | 10 074 720 | 910 | 11 026 932 | 24 660 513 |
| vasy_1112_5290 | 160 061 | 71 968 | 1 112 490 | 5 290 860 |
| vasy_12323_27667 | 11 214 774 | 910 | 12 323 703 | 27 667 803 |
| vasy_2581_11442 | 274 690 | 26 796 | 2 581 374 | 11 442 382 |
| vasy_4220_13944 | 2398 982 | 49 151 | 4 220 790 | 13 944 372 |
| vasy_4338_15666 | 828 412 | 26 796 | 4 338 672 | 15 666 588 |
| vasy_6020_19353 | 2 041 | 6 013 920 | 6 020 550 | 19 353 474 |
| vasy_6120_11031 | 4 638 059 | 1902 | 6 120 718 | 11 031 292 |
| vasy_8082_42933 | 323 629 | 7 054 752 | 8 082 905 | 42 933 110 |
| swp_d2w2q2.s | 1 | 1 429 676 | 1 429 676 | 6 704 544 |
| swp_d2w2q3.s | 1 | 5 323 836 | 5 323 836 | 25 236 056 |
| swp_d3w2q2.s | 1 | 5 168 596 | 5 168 596 | 24 615 576 |

space complexity is not a problem of OBFR. However, the FB algorithm exceeded depth 200 in our experiments. It did not prevent the algorithm from successful computation of SCCs, because our graphs are relatively small. Nevertheless, this high-recursion depth kills the benefit of having accumulated memory of a cluster of workstations. If we add that FB is much slower if independent sub-graphs are not processed in parallel, we can conclude that FB is not a very good distributed algorithm. On the other hand, OBF + FB reached maximum recursion depth of 17. It seems that the

TABLE 3. Run-times for synthetic graphs (in seconds)

| State space | FB-P | OBFR-S | OBFR-P | OBFR-MP | OBF-FB-P | CH |
|---|---|---|---|---|---|---|
| L10L10T10 | 10 | 128 | 25 | 8 | 8 | 75 |
| L100L100T4 | 13 | 19 | 13 | 11 | 5 | 145 |
| L15L15T10 | 16 | 118 | 56 | 16 | 17 | 142 |
| L4L4T16 | 2743 | 6603 | 671 | 309 | 297 | 325 |
| L20L20T12 | 224 | 575 | 287 | 74 | 71 | 456 |
| L80L80T8 | 94 | 107 | 110 | 34 | 45 | 795 |
| L350L350T4 | 83 | 91 | 88 | 38 | 45 | 1583 |
| L1750L1750T0 | 34 | 31 | 43 | 17 | 16 | 1021 |
| L1750L1750T1 | 148 | 138 | 166 | 87 | 82 | 6533 |
| Li200Lo10 | 1982 | 1964 | 1131 | 76 | 58 | 9317 |
| Li125Lo16 | 1105 | 975 | 740 | 61 | 52 | 5827 |
| Li100Lo20 | 754 | 588 | 520 | 65 | 51 | 4513 |
| Li80Lo25 | 548 | 465 | 454 | 57 | 77 | 3560 |
| Li67Lo30 | 510 | 356 | 484 | 58 | 44 | 3080 |
| Li50Lo40 | 357 | 236 | 163 | 48 | 48 | 3350 |
| Li40Lo50 | 286 | 175 | 126 | 50 | 43 | 2628 |
| Li30Lo67 | 174 | 127 | 110 | 43 | 44 | 2364 |
| Li25Lo80 | 140 | 102 | 103 | 46 | 46 | 2972 |
| Li20Lo100 | 176 | 88 | 80 | 43 | 40 | 2782 |
| Li16Lo125 | 106 | 77 | 115 | 71 | 38 | 2148 |
| Li10Lo200 | 81 | 58 | 90 | 62 | 45 | 1895 |

TABLE 4. Runtimes for real graphs (in seconds)

| State space | FB-P | OBFR-S | OBFR-P | OBFR-MP | OBF-FB-P | CH |
|---|---|---|---|---|---|---|
| cwi_2165_8723 | 21 | 43 | 30 | 29 | 22 | 49 |
| cwi_2416_17605 | 76 | 8791 | 942 | 51 | 56 | 126 |
| cwi_7838_59101 | 65 | 58 | 107 | 102 | 72 | 227 |
| vasy_11026_24660 | 3387 | n/a | 3391 | 416 | 827 | 471 |
| vasy_1112_5290 | 168 | 5611 | 399 | 73 | 73 | 365 |
| vasy_12323_27667 | 4483 | n/a | 3942 | 500 | 1016 | 509 |
| vasy_2581_11442 | 169 | 6182 | 2084 | 64 | 109 | 276 |
| vasy_4220_13944 | 531 | 8348 | 976 | 347 | 1987 | 151 |
| vasy_4338_15666 | 209 | 14352 | 4445 | 107 | 110 | 310 |
| vasy_6020_19353 | 60 | 147 | 93 | 51 | 34 | 130 |
| vasy_6120_11031 | 888 | 26611 | 1483 | 282 | 299 | 592 |
| vasy_8082_42933 | 162 | 440 | 640 | 455 | 407 | 280 |
| swp_d2w2q2.s | 12 | 9 | 12 | 16 | 6 | 44 |
| swp_d2w2q3.s | 55 | 13 | 28 | 55 | 18 | 102 |
| swp_d3w2q2.s | 38 | 16 | 42 | 35 | 15 | 70 |
| Total run-time | 10324 | >142621 | 18572 | 2583 | 5051 | 3702 |

uppermost OBF is so successful in slicing the whole graph, that the amount of work left for FB that processes the slices is relatively small.

And now for some comments on the measured run-times. First, for the synthetic graphs. As one can see from Table 3, OBFR-MP and OBF-FB-P together are clear winners. Their run-times are practically the same because most of the decomposition was done by the first OBF which is the same

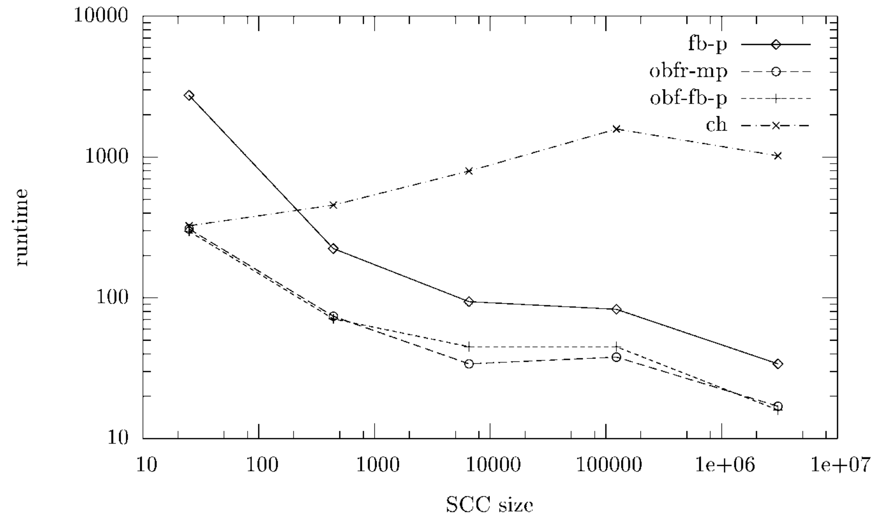


FIGURE 11. Dependency of run-time on SCC size, type 1 synthetic graphs (log. scale)

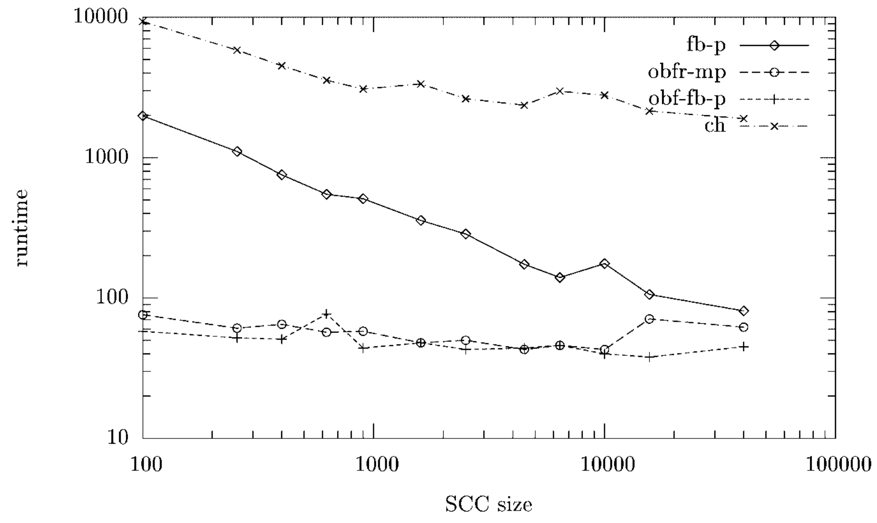


FIGURE 12. Dependency of run-time on SCC size, type 2 synthetic graphs (log. scale)

for both algorithms. The slices identified by the OBF were then processed in parallel. It did not matter if OBF or FB was used for them because of the structure of the slices.

FB, OBFR-S and OBFR-P worked quite well on graphs with large SCCs, but they require a long time to decompose a graph with many small components. OBFR-P was the best of them, but its performance on graphs with many small components is still poor. The reason for the big difference between OBFR-P and OBFR-MP is that some slices identified by the first OBF contained many parts with no edges between them and waiting for OBF to finish on one part before moving to next part affects the performance considerably.

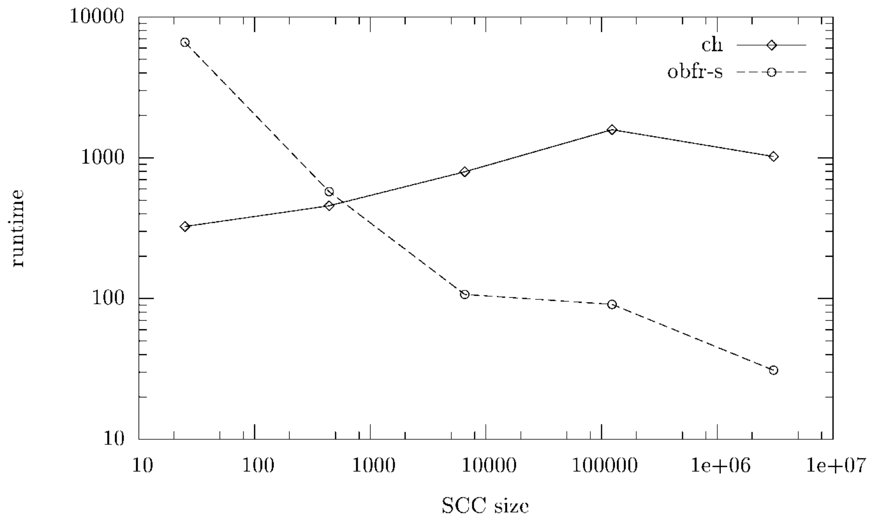


FIGURE 13. Dependency of run-time on SCC size, comparison of OBFR-S and CH, type 1 synthetic graphs (log. scale)

Interestingly enough, for the synthetic graphs of type 1, unlike most of the other algorithms, especially OBFR-S, the CH algorithm worked better on graphs with many small components (Figure 13). We attribute this to the high degree parallelism in CH which outweighs the extra costs due to re-colouring in this case. However, it was not confirmed on type 2 graphs (Figure 12), on which CH has extremely poor performance. This is explained by many paths of the same length leading to the same vertex, which causes frequent re-colouring.

The experiments on real graphs (Table 4) have only one winner, OBFR-MP. Yet, its victory was not as clear as the victory for synthetic graphs. In particular, CH turned out to be successful. We included total run-times for all real graphs to allow for better comparison.

The structure of the graphs was not regular, so OBFR had to go deeper to decompose the graph. Since the decomposition was not done by the first OBF, the FB algorithm had much more work in OBF + FB than for synthetic graphs, which resulted in poor behaviour for some graphs, especially vasy_12323_27667 and vasy_4220_13944.

## 6 Conclusion

In this article, we listed and compared known distributed algorithms for the decomposition of directed graphs into their SCCs. We also proposed a new algorithm, called OBFR, based on recursive application of the OBF technique introduced in [4]. The correctness of the new algorithm was proven formally. We also report on an extensive experimental study we did to evaluate the new algorithm. OBFR outperformed all the other algorithms in most cases.

Our experiments show that the way the algorithm is implemented influences its performance a great deal. In particular, the best implementation turned out to be the one with the highest degree of parallelism, that is the one which starts another parallel procedure every time a part of the graph that can be processed independently has been identified.

There is one type of graphs where the CH algorithm [22] may be the best choice. These are graphs consisting of many unconnected islands. Such graphs arise for instance when considering only (invisible) $\tau$-transitions as a pre-processing step to branching bisimulation reduction. CH starts working on all islands simultaneously, but all the other algorithms process them one by one unless they contain indegree 0 vertices. If these islands are small enough, re-colouring is not a problem and CH is very fast. This suggests an aim for future work: to improve OBFR to work better on graphs with many unconnected islands.

OBFR is also suitable for multi-core shared-memory architectures that are going to be the standard in the near future. Implementing and evaluating OBFR on such architectures is another aim for future work.

## Acknowledgement

We like to thank Simona Orzan for discussions on the CH-algorithm and sharing her implementation. We are grateful to the Centrum voor Wiskunde en Informatica (Amsterdam) for hosting a considerable part of the research reported here.

## Funding

Czech Science Foundation (grant No. 201/09/1389); Academy of Sciences (grant No. 1ET408050503); EU FP6 NEST pathfinder project EC-MOAN (043235).

## References

[1] N. Amato. Improved processor bounds for parallel algorithms for weighted directed graphs. *Information Processing Letters*, **45**, 147–152, 1993.

[2] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai and P. Šimeček. Divine – a tool for distributed verification. In *Proceedings of Computer Aided Verification*, Vol. 4144/2006 of *Lecture Notes in Computer Science*, pp. 278–281. Springer Berlin/Heidelberg, 2006.

[3] J. Barnat, J. Chaloupka and J. C. van de Pol. Improved distributed algorithms for SCC decomposition. In *Participant proceedings of the Sixth International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007)*, pp. 65–80. CTIT, University of Twente, 2007.

[4] J. Barnat and P. Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. In *Proceedings of the 5th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2006)*. Vol. 4346 of *Lecture Notes in Computer Science*. pp. 316–330. Springer-Verlag, 2007.

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

[6] S. C. C. Blom, J. R. Calamé, B. Lisser, S. Orzan, J. Pang, J. C. van de Pol, M. Torabi Dashti and A. J. Wijs. Distributed analysis with $\mu$crl: a compendium of case studies. In *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, eds. Vol. 4424 of *Lecture Notes in Computer Science*, pp. 683–689. Springer Verlag, Braga, Portugal, Berlin, 2007.

[7] S. C. C. Blom and H. Garavel. The VLTS benchmark suite. Available at: `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`, 2003. (last accessed on 4 February, 2009).

[8] S. C. C. Blom and J. C. van de Pol. State space reduction by proving confluence. In *CAV*, E. Brinksma and K. G. Larsen, eds. Vol. 2404 of *Lecture Notes in Computer Science*, pp. 596–609. 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 *Proceddings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*. Vol. 3312 of *Lecture Notes in Computer Science*, pp. 352–366. Springer-Verlag, 2004.

[10] S. Caselli, G. Conte and P. Marenzoni. Parallel state space exploration for GSPN models. In *Applications and Theory of Petri Nets 1995*, G. De Michelis and M. Diaz, eds. Vol. 935 of *Lecture Notes in Computer Science*, pp. 181–200. Springer-Verlag, 1995.

[11] G. Ciardo, J. Gluckman and D.M. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal of Computing*, **47**, 153–167, 1997.

[12] F. Ciesinski and C. Baier. LiQuor: a tool for qualitative and qualitative linear time analysis of reactive systems, 3rd International Conference on the Quantitative Evaluation of Systems (QEST 2006), 131-132, *IEEE Computer Society*, 2006.

[13] E. M. Clarke, O. Grumberg and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[14] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, **81**, 334–352, 1989.

[15] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[16] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of the Tools and Algorithms for Construction and Analysis of Systems*. Vol. 2031 of *Lecture Notes in Computer Science*, pp. 420–434. Springer, 2001.

[17] L. K. Fleischer, B. Hendrickson and A. Pinar. On identifying strongly connected components in parallel. Vol. 1800 of *Lecture Notes in Computer Science*, Springer, pp. 505–511, 2000.

[18] H. Garavel, R. Mateescu and I. M. Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*. Vol. 2057 of *Lecture Notes in Computer Science*, pp. 200–216. Springer-Verlag, 2001.

[19] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, **28**, 61–65, 1988.

[20] W. McLendon III, B. Hendrickson, S. J. Plimpton and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel Distribution Computation*, **65**, 901–910, 2005.

[21] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*. Vol. 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

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

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

[24] U. Stern and D. L. Dill. Parallelizing the Mur$\varphi$ verifier. In *Proceedings of Computer Aided Verification (CAV '97)*. O. Grumberg, ed., Vol. 1254 of *Lecture Notes in Computer Science*, pp. 256–267. Springer-Verlag, 1997.

[25] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on computing*, **1**, pp. 146–160, 1972.

Received 29 February 2008

# I/O Efficient Accepting Cycle Detection*

Jiri Barnat, Lubos Brim, and Pavel Šimeček

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

**Abstract.** We show how to adapt an existing non-DFS-based accepting cycle detection algorithm OWCTY [10,15,29] to the I/O efficient setting and compare its I/O efficiency and practical performance to the existing I/O efficient LTL model checking approach of Edelkamp and Jabbar [14]. The new algorithm exhibits similar I/O complexity with respect to the size of the graph while it avoids quadratic increase in the size of the graph. Therefore, the number of I/O operations performed is significantly lower and the algorithm exhibits better practical performance.

## 1 Introduction

Model checking became one of the standard technique for verification of hardware and software systems even though the class of systems that can be fully verified is fairly limited due to the well known *state explosion problem* [12]. The automata-theoretic approach [33] to model checking finite-state systems against linear-time temporal logic (LTL) reduces to the detection of reachable accepting cycles in a directed graph. Due to the state explosion problem, the graph tends to be extremely large and its size poses real limitations to the verification process. Many more-or-less successful techniques have been introduced [12] to reduce the size of the graph advancing thus the frontier of still tractable systems. Nevertheless, for real-life industrial systems these techniques are not efficient enough to fit the data into the main memory. An alternative solution is to increase the computational resources available to the verification process. The two major approaches include the usage of clusters of workstations and the usage of external memory devices (disks).

Regarding external memory devices, the goal is to develop algorithms that minimize the number of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on an external device is orders of magnitude slower than the access to information stored in the main memory. Thus the complexity of I/O efficient algorithms is measured in the number of I/O operations [1].

A lot of effort has been put into research on I/O efficient algorithms working on explicitly stored graphs [11,20,24,25]. For an explicitly stored graph, an I/O efficient algorithm typically has to perform a random access operation every

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 281–293, 2007.

time it needs to enumerate edges incident with a given vertex. However, in model checking, the graphs are often given implicitly which means that the edges incident with a given vertex are computed on demand from the vertex itself. Thus, an algorithm working on an implicitly given graph may save up to $|V|$ random access operations, which may have significant impact on the performance of the algorithm in practice.

A distinguished technique that allows for an I/O efficient implementation of a graph traversal procedures is the so called *delayed duplicate detection* [21,22,26,32]. A traversal procedure has to maintain a set of visited vertices to prevent their re-exploration. Since the graphs are large, the set cannot be completely kept in the main memory and must be stored on the external memory device. When a new vertex is generated it is checked against the set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them together in a group for the price of a single scan operation.

Unfortunately, the delayed duplicate detection technique is incompatible with the depth-first search (DFS) of a graph [14]. Therefore, most approaches to I/O efficient (LTL) model checking suggested so far, have focused on the state space generation and verification of safety properties only. The first I/O efficient algorithm for state space generation has been implemented in Mur$\varphi$ [32]. Later on, several heuristics for the state space generation were suggested and implemented in various verification tools [16,18,23]. The first attempt to verify more than safety properties was described in [19], however, the suggested approach uses the random search to find a counterexample to a given property. Therefore, it is incomplete in the sense that it is not able to prove validity of the property.

To the best of our knowledge, the only *complete* I/O efficient LTL model checker was suggested by Edelkamp and Jabbar in [14] where the problematic DFS-based algorithm was avoided by the reduction of the accepting cycle detection problem to the reachability problem [7,31] whose I/O efficient solution was further improved by using the directed ($A^*$) search and parallelism. The algorithm works in the on-the-fly manner meaning that only a part of the state space is constructed, which is needed in order to check the desired property. The reduction transforms the graph so that the size of the graph after the transformation is asymptotically quadratic with respect to the original one. More precisely, the size of the resulting graph is $|F| \times |G|$, where $|G|$ is the size of the original graph and $|F|$ is the number of accepting vertices. As the external memory algorithms are meant to be applied to large scale graphs, the quadratic increase in the size of the graph is significant and, according to our experience, it often aborts due to the lack of space. This is especially the case when the model is valid and the entire graph has to be traversed to prove the absence of an accepting cycle. The approach is thus mainly useful for finding counterexamples in the case a standard verification tool fails due to the lack of memory. However, completeness is a very important aspect of LTL model checking as well. A typical scenario is that if the system is invalid and the counterexample found, the system is corrected and the property verified again. In the end, the graph must be traversed completely anyway.

Since DFS-based algorithms cannot be used for I/O efficient solution to the accepting cycle detection, a non-DFS algorithm is required. The situation very much resembles a similar one encountered in cluster-based approach to LTL model checking [2]. The main problem of the approach is that the optimal sequential algorithm (e.g. Nested DFS [17]) is inherently sequential and hence difficult to be parallelized [30]. Consequently, several new parallel algorithms that do not build on top of the depth-first search have been introduced [3,4,8,9,10].

In this paper we show how to adapt a parallel enumerative version of the *One Way Catch them Young Algorithm* (OWCTY) [10,15,29] to the I/O efficient setting and compare its I/O efficiency and practical performance with the I/O efficient LTL model checking algorithm by Edelkamp and Jabbar [14].

## 2 I/O Efficient OWCTY Algorithm

As discussed above, an I/O efficient solution to LTL model checking has to build upon a non-DFS algorithm. A particularly suitable algorithm for enumerative LTL model checking was described in [10]. The goal of the algorithm is to compute the set of vertices that are reachable from a vertex on an accepting cycle. If the set is empty, there is no accepting cycle in the graph, otherwise the presence of an accepting cycle is ensured [15,29].

The algorithm repeatedly computes approximations of the target set until a fixpoint is reached. All reachable vertices are inserted into the approximation set (*ApproxSet*) within the procedure Initialize-ApproxSet. After that, vertices violating the condition are gradually removed from the approximation set using procedures Elim-No-Accepting and Elim-No-Predecessors. Procedure Elim-No-Accepting removes those vertices from the approximation set that have no accepting ancestors in the set, i.e. vertices that lie on leading non-accepting cycles. Procedure Elim-No-Predecessors removes vertices that have no ancestors at all, i.e. leading vertices lying outside a cycle. The pseudo-code is given as Algorithm 1.

**Algorithm 1.** DetectAcceptingCycle

```
Require: Implicit definition of G=(V,E,ACC)
1: Initialize-ApproxSet()
2: $oldSize \leftarrow \infty$
3: while $(ApproxSet.size \neq oldSize) \wedge (ApproxSet.size > 0)$ do
4:     $oldSize \leftarrow ApproxSet.size$
5:     Elim-No-Accepting()
6:     Elim-No-Predecessors()
7: return $ApproxSet.size > 0$
```

The approximation set induces an approximation graph. The in-degree of a vertex in the approximation graph corresponds to the number of its immediate predecessors in the approximation set. To identify vertices without ancestors in the approximation set, the in-degree is maintained for every vertex of the approximation graph. Procedure Elim-No-Predecessors then works as follows. All vertices from the set with a zero in-degree are moved to a queue from where they are dequeued one by one. Dequeued vertices are eliminated from the set, and the in-degrees of its descendants are updated. If an in-degree drops to zero, the corresponding vertex is inserted into the queue to be eliminated as well. The procedure eliminates vertices in a topological order and hence the queue becomes empty as soon as all vertices preceding a cycle are eliminated.

Procedure Elim-No-Accepting works as follows. If a vertex has an accepting ancestor in the approximation set, it has to be reachable from some accepting vertex in the set. Therefore, the procedure first removes all non-accepting vertices from the set and sets the numbers of predecessors of all vertices remaining in the set to zero. Then a forward search is performed starting from the vertices remaining in the set. During the search all visited vertices are re-inserted to the approximation set and the numbers of immediate predecessors of vertices in the set are re-counted.

There are three major data structures used by the algorithm. These are *Candidates*, *ApproxSet*, and *Open*. *Candidates* is the set of vertices strictly kept in memory that is used for the delayed duplicate detection technique. It keeps vertices that have been processed and are waiting to be checked against the set of vertices stored on the external device. *ApproxSet* is the set of vertices belonging to the current approximation set. It is implemented as a linear list and stored externally. Together with *Candidates*, it is used as the set of vertices already visited during the forward exploration of the graph in procedure Elim-No-Accepting. For that purpose, both *Candidates* and *ApproxSet* data structures are modified to keep not only vertices, but also the corresponding numbers of relevant immediate predecessors. The number associated with a particular vertex $s$ is referred to as the *appendix* of the vertex and is set and read with methods *setAppendix(s)* and *getAppendix(s)*, respectively. Finally, the data structure *Open* is a queue of vertices. It is used to keep open vertices during the breadth-first exploration of the graph within procedure Elim-No-Accepting, and vertices to be eliminated (vertices without any predecessors) during the execution of procedure Elim-No-Predecessors. The data structure *Open* is stored in the external memory, the vertices are, however, inserted into and taken from it in a strict FIFO manner. Thus, a possible I/O overhead could be minimized using an appropriate buffering mechanism.

In some of its phases, the algorithm performs a *scan* through the externally stored set of vertices (*ApproxSet*) and decides about every vertex if it should be removed from the set or not. To preserve the I/O efficiency of such an operation, a temporary external data structure *ApproxSet'* is introduced. In particular, vertices that should remain in the set are copied to the temporary structure.

**Algorithm 2.** MERGE

```
 1: if $mode$ = Elim-No-Accepting then
 2:     for all $s \in ApproxSet$ do
 3:         if $s \in Candidates$ then
 4:             $app \leftarrow Candidates.getAppendix(s)$
 5:             $app' \leftarrow ApproxSet.getAppendix(s)$
 6:             $Candidates \leftarrow Candidates \setminus \{s\}$
 7:             $ApproxSet.setAppendix(s, app + app')$
 8:     for all $s \in Candidates$ do
 9:         $Open.pushBack(s)$
10:         $ApproxSet \leftarrow ApproxSet \cup \{s\}$
11: else
12:     $ApproxSet' \leftarrow \emptyset$
13:     for all $s \in ApproxSet$ do
14:         $app' \leftarrow ApproxSet.getAppendix(s)$
15:         if $s \in Candidates$ then
16:             $app \leftarrow Candidates.getAppendix(s)$
17:             if $(app + app') = 0$ then
18:                 $Open.pushBack(s)$
19:             else
20:                 $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
21:                 $ApproxSet'.setAppendix(s, app + app')$
22:         else
23:             $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
24:             $ApproxSet'.setAppendix(s, app')$
25:     $ApproxSet \leftarrow ApproxSet'$
26: $Candidates \leftarrow \emptyset$
```

Once the scan is complete, the content of the original *ApproxSet* is discarded and replaced with the content of the temporary structure *ApproxSet'*.

Having described the data structures we are ready to introduce several auxiliary subroutines. The most important one is procedure MERGE that is responsible for merging information about vertices stored in the internal memory (*Candidates*) and vertices stored externally (*ApproxSet*). The procedure can operate in two different modes according to the value of the variable *mode*. The two modes correspond to the top most procedures ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS. In the mode **Elim-No-Accepting**, vertices from set *Candidates* are merged with vertices from *ApproxSet* and the result is stored externally to *ApproxSet*. For already visited vertices the corresponding appendices are just combined and stored externally. Moreover, newly discovered vertices are inserted into the queue of vertices to be further processed (*Queue*). In the mode **Elim-No-Predecessors**, no new vertices are discovered, hence only the appendices are combined. Vertices with zero in-degree are removed from the external memory and in-degree of their immediate descendants is appropriately decreased. For the details see Algorithm 2.

**Algorithm 3.** StoreOrCombine

```
Require: $s$, $app$
1: if $s \in Candidates$ then
2:     $app' \leftarrow Candidates.getAppendix(s)$
3:     $Candidates.setAppendix(s,\ app{+}app')$
4: else
5:     $Candidates \leftarrow Candidates \cup \{s\}$
6:     $Candidates.setAppendix(s, app)$
7:     if MemoryIsFull() then
8:         Merge()
```

Another auxiliary procedure is procedure StoreOrCombine whose purpose is to insert a vertex into the candidate set if the vertex is not yet present in the set, or update the corresponding appendix of the vertex, otherwise. Once the main memory becomes full, vertices from the candidate set are processed and the candidate set is emptied by procedure Merge.

**Algorithm 4.** OpenIsNotEmpty

```
1: if $Open.isEmpty()$ then
2:     Merge()
3: return $\neg Open.isEmpty()$
```

The last auxiliary function is a function for checking the emptiness of the queue of vertices to be processed (*Open*). If the queue is empty, procedure OpenIsNotEmpty calls procedure Merge to perform the delayed duplicate detection. The procedure returns **False**, if *Open* is empty and merging has not brought any new vertices to be processed.

Algorithm 5 and Algorithm 6 give pseudo-codes of the two main procedures. Note that algorithm DetectAcceptingCycle uses functions GetInitialVertex, GetSuccessors, and IsAccepting to traverse the graph and to check whether a vertex is accepting or not. These functions are part of the implicit definition of the graph. Procedure Elim-No-Accepting has actually two goals. First, to eliminate those vertices from the approximation set that are unreachable from accepting vertices in the set, and second, to properly count the in-degrees in the approximation graph. Procedure Elim-No-Predecessors employs the in-degrees to recursively remove vertices without predecessors from the approximation set.

An important observation is that it is not necessary to initialize the approximation set with the set of all vertices. Since the first procedure in the very first iteration of the while loop performs forward exploration of the graph starting from accepting vertices in the set, it is enough to initialize the set with "leading" accepting vertices only, i.e. those accepting vertices that have no accepting ancestors. Such vertices can be identified with a simple forward traversal that is

**Algorithm 5.** Elim-No-Accepting

```
 1: $mode \leftarrow$ **Elim-No-Accepting**
 2: $ApproxSet' \leftarrow \emptyset$
 3: for all $s \in ApproxSet$ do
 4:     if IsAccepting($s$) then
 5:         $Open.pushBack(s)$
 6:         $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
 7:         $ApproxSet'.setAppendix(s, 0)$
 8: $ApproxSet \leftarrow ApproxSet'$
 9: while OpenIsNotEmpty() do
10:     $s \leftarrow Open.popFront()$
11:     for all $t \in$ GetSuccessors($s$) do
12:         StoreOrCombine($t, 1$)
```

**Algorithm 6.** Elim-No-Predecessors

```
 1: $mode \leftarrow$ **Elim-No-Predecessors**
 2: $ApproxSet' \leftarrow \emptyset$
 3: for all $s \in ApproxSet$ do
 4:     if $ApproxSet.getAppendix(s) = 0$ then
 5:         $Open.pushBack(s)$
 6:     else
 7:         $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
 8: $ApproxSet \leftarrow ApproxSet'$
 9: while OpenIsNotEmpty() do
10:     $s \leftarrow Open.popFront()$
11:     for all $t \in$ GetSuccessors($s$) do
12:         StoreOrCombine($t, -1$)
```

**Algorithm 7.** Initialize-ApproxSet

```
 1: $mode \leftarrow$ **Elim-No-Accepting**
 2: $Candidates \leftarrow \emptyset$
 3: $s \leftarrow$ GetInitialVertex()
 4: $ApproxSet \leftarrow \{s\}$
 5: if $\neg$ IsAccepting($s$) then
 6:     $Open.pushBack(s)$
 7:     while OpenIsNotEmpty() do
 8:         $s \leftarrow Open.popFront()$
 9:         for all $t \in$ GetSuccessors($s$) do
10:             if IsAccepting($t$) then
11:                 $ApproxSet \leftarrow ApproxSet \cup \{t\}$
12:             else
13:                 StoreOrCombine($t, 0$)
```

allowed to explore descendants of non-accepting vertices only. See the pseudocode given as Algorithm 7.

## 3 Complexity Analysis

A widely accepted model for the analysis of the complexity of I/O algorithms is the model of Aggarwal and Vitter [1], where the complexity of an I/O algorithm is measured in terms of the numbers of external I/O operations only. This is motivated by the fact that a single I/O operation is by approximately six orders of magnitude slower than a computation step performed in the main memory [34]. Therefore, an algorithm that does not perform the optimal amount of work but has a lower I/O complexity, may be faster in practice compared to an algorithm that performs the optimal amount of work, but has a higher I/O complexity. The complexity of an I/O algorithm in the model of Aggarwal and Vitter is further parametrized by $M$, $B$, and $D$, where $M$ denotes the number of items that fits into the internal memory, $B$ denotes the number of items that can be transferred in a single I/O operation, and $D$ denotes the number of blocks that can be transferred in parallel, i.e. the number of independent parallel disks available. The abbreviations $sort(n)$ and $scan(n)$ stand for $\theta(N/(DB)log_{M/B}(N/B))$ and $\theta(N/(DB))$, respectively. In this section we give the I/O complexity of our algorithm and compare it with the complexity of the algorithm from [14].

We use the following notation. *BFS tree* is a tree given by the graph traversal from the initial set of vertices in the breadth-first order. Its height $h_{BFS}$ is called *BFS height*, its levels are called *BFS levels*. *SCC graph* is a directed acyclic graph, whose vertices are maximal strongly connected components of the graph and the edges are given according to the reachability relation between the components. Let $l_{SCC}$ denote the length of the longest path in the SCC graph. The I/O complexity of the algorithm is given in Theorem 1. The proof of the complexity can be found in the full version of the paper [6].

**Theorem 1.** *The I/O complexity of algorithm* DetectAcceptingCycle *is*

$$\mathcal{O}(l_{SCC} \cdot (h_{BFS} + |p_{max}| + |E|/M) \cdot scan(|V|)),$$

*where $p_{max}$ is the longest path in the graph going through trivial strongly connected components (without self-loops).*

For the purpose of comparison we denote our new algorithm as $DAC$ and the algorithm of Edelkamp and Jabbar [14] as *EJ*. Theorem 1 of [14] claims that *EJ* is able to detect accepting cycles with I/O complexity $\mathcal{O}(sort(|F||E|) + l \cdot scan(|F||V|))$, where $|F|$ is the number of accepting states and $l$ is the length of the shortest counterexample.

The complexity of *EJ* is not easy to compare with our results, because the two algorithms use different ways to maintain the set of candidates. The candidate set can be either stored externally ($EJ$) or internally ($DAC$). In the case that the candidate set is stored externally, it is possible to perform the merge operation on a BFS level independently of the size of the main memory. Therefore, this approach is suitable for those cases where memory is small or the graph is by orders of magnitude larger. The disadvantage of the approach is that it needs *sort* operations and it cannot be combined with heuristics, such as bit-state hashing and a lossy hash table [16]. Fortunately, both *EJ* and *DAC* are modular enough to be able to work in both modes. Table 1 gives I/O complexities of all four variants, where *EJ'* denotes algorithm *EJ* modified so that the candidate set is kept in the internal memory, and *DAC'* denotes algorithm *DAC* modified so that the candidate set is stored externally.

**Table 1.** I/O complexity of algorithms for both modes of storage of the candidate set

Candidate set in the main memory:

| | |
|---|---|
| EJ' | $\mathcal{O}((l + \|F\|\|E\|/M) \cdot scan(\|F\|\|V\|))$ |
| DAC | $\mathcal{O}(l_{SCC} \cdot (h_{BFS} + \|p_{max}\| + \|E\|/M) \cdot scan(\|V\|))$ |

Candidate set in the external memory:

| | |
|---|---|
| EJ | $\mathcal{O}(l \cdot scan(\|F\|\|V\|) + sort(\|F\|\|E\|))$ |
| DAC' | $\mathcal{O}(l_{SCC} \cdot ((h_{BFS} + \|p_{max}\|) \cdot scan(\|V\|) + sort(\|E\|)))$ |

In the worst case the values of $l_{SCC}$, $|p_{max}|$, and $h_{BFS}$ are equal to $|V|$. Thus the worst case I/O complexity of *DAC* is better than that of *EJ'* and the worst case I/O complexity of *DAC'* is equal to that of *EJ*, provided that $l = |V|$ and $|F| = |V|$.

Note that for graphs of verified systems the numbers $l_{SCC}$, $|p_{max}|$, and $h_{BFS}$ are typically smaller by several orders of magnitude than the number of vertices. $l_{SCC}$ (giving the upper bound to the number of iterations of the loop of Algorithm 1) usually ranges from 1 to 20 [15]. $h_{BFS}$ is not proportional to the size of the state space and oscillates around several hundreds [27], so the $|p_{max}|$ according to our own measurements. However, the number of accepting vertices ($F$) is quite often in the same order of magnitude as the number of vertices. Therefore, *EJ'* and *EJ* suffer from the graph blow-up and perform much more I/O operations compared to *DAC* and *DAC'*, respectively. On the other hand, *EJ'* and *EJ* work on-the-fly and can thus outperform *DAC* and *DAC'* on the graphs with small number of accepting vertices and short counterexamples. Nevertheless, short counterexamples are also easy to find using on-the-fly internal memory model checkers which outperform both external memory approaches.

Regarding space complexity, *DAC* is more space efficient than *EJ*. Since *EJ'* needs to remember all visited pairs of vertices, where a pair consists of one accepting and one arbitrary vertex, the space complexity of the algorithm is $\mathcal{O}(|F||V|)$, i.e. asymptotically quadratic in the size of the graph. On the other hand, the space complexity of *DAC* is $\mathcal{O}(|V|)$, as it only maintains the approximation set, queue and the candidate set whose sizes are always bounded by the number of vertices. The same holds for the pair *EJ* and *DAC'*.

## 4 Experimental Evaluation

In order to obtain experimental evidence about how our algorithm behaves in practice, we implemented both algorithms and compared them mutually as well as with the model checker SPIN with all the default reduction techniques (including partial order) turned on.

Algorithm *DetectAcceptingCycle* (*DAC*) has been implemented upon DiVinE Library [5], providing the state space generator, and STXXL Library [13], providing the necessary I/O primitives. Algorithm *EJ* was implemented as a procedure that performs the graph transformation as suggested in [14] and then employs I/O efficient breadth-first search to check for the counterexample. Note that our implementation of [14] does not have the $A^*$ heuristics and so it can be less efficient in the search for the counterexample. The procedure is referred to as *Liveness as Safety with BFS* (*LaS-BFS*).

We have measured run times and a memory consumption of SPIN, *LaS-BFS* and *DAC* on a collection of systems and their LTL properties taken from the BEEM project [28]. The models were selected so that the state spaces generated by SPIN and DiVinE were exactly of the same size. The experimental results are listed in Table 2. Note that just before the unsuccessful termination of *LaS-BFS* due to exhausting the disk space the size of BFS levels exhibited growing

**Table 2.** Run times and memory consumption on a single workstation with 2 GB of RAM and 60 GB of available hard disk space. The time is given in `hh:mm:ss` format.

| | | SPIN | | LaS-BFS | | DAC | |
|---|---|---|---|---|---|---|---|
| | States | Time | RAM | Time | Disk | Time | Disk |
| Phils(16,1),P3 | 61,230,206 | Out of memory | | Out of disk space | | 02:01:11 | 5.5 GB |
| MCS(5),P4 | 119,663,657 | Out of memory | | Out of disk space | | 03:32:41 | 8 GB |
| Szymanski(5),P4 | 419,183,762 | Out of memory | | Out of disk space | | 44:49:36 | 32 GB |
| Elevator2(16),P4 | 76,824,540 | Out of memory | | Out of disk space | | 11:37:57 | 9.2 GB |
| Leader Fil.(7),P2 | 431,401,020 | 00:01:35 | 1369 MB | Out of disk space | | 32:03:52 | 42 GB |

Valid properties on large models.

| | | SPIN | | LaS-BFS | | DAC | |
|---|---|---|---|---|---|---|---|
| | States | Time | RAM | Time | Disk | Time | Disk |
| Lamport(3),P4 | 56,377 | 00:00:01 | 18 MB | 00:55:34 | 799 MB | 00:00:19 | 6,1 MB |
| Anderson(4),P2 | 58,205 | 00:00:01 | 20 MB | 00:11:11 | 153 MB | 00:00:18 | 6,1 MB |
| Peterson(4),P4 | 2,239,039 | 00:00:08 | 85 MB | Out of disk space | | 00:04:44 | 159 MB |

Valid properties on small models.

| | | SPIN | | LaS-BFS | | DAC | |
|---|---|---|---|---|---|---|---|
| | States | Time | RAM | Time | Disk | Time | Disk |
| Bakery(5,5),P3 | 506,246,410 | 00:00:01 | 16 MB | 01:34:13 | 5,4 GB | 69:27:58 | 38 GB |
| Szymanski(4),P2 | 4,555,287 | 00:00:01 | 18 MB | 00:59:00 | 203 MB | 00:19:55 | 205 MB |
| Elevator2(7),P5 | 43,776 | 00:00:01 | 17 MB | 00:01:15 | 121 MB | 00:00:18 | 6,1 MB |

Invalid properties.

tendency. This suggests that the computation would last substantially longer if sufficient disk space was available. For the same input graphs, algorithm *DAC* manage to perform the verification using a few GBs of space only.

Measurements on large systems with valid formulas demonstrate that *DAC* is able to successfully prove the correctness of systems, on which SPIN and *LaS-BFS* fail. However, there are systems and valid formulas, which take a long time to verify by our algorithm, but can be verified quickly using SPIN (e.g. model *Leader Filters*). This is due to the partial order reduction technique, which is extraordinarily efficient in this case. Results on small systems show the state space blow-up in case of *LaS-BFS*. E.g. on the model *Lamport*, 6,1 MB of disk space is enough for *DAC* to store the entire state space while *LaS-BFS* needs 799 MB. As for systems with invalid formulas, the new algorithm is slow, since it does not work on-the-fly. Nevertheless, it is able to finish if the state space fits in the external memory. Moreover, it is faster than *LaS-BFS* on systems with long counterexamples as the space space blow-up takes effect when *LaS-BFS* has to traverse a substantial part of the state space (e.g. model *Elevator2*).

In summary, the new algorithm is especially useful for verification of large systems with valid formulas where SPIN fails due to the limited size of the main memory and *LaS-BFS* runs out of the available external memory because of a large amount of accepting states. On systems with invalid formulas, algorithm *DAC* finishes if the state space fits in the external memory, but it may take quite a long time as it does not work on-the-fly.

## 5 Conclusions and Future Work

In this paper we presented a new I/O efficient algorithm for accepting cycle detection on implicitly given graphs. The algorithm exhibits linear space complexity while preserving practically reasonable I/O complexity. Another indirect contribution of the paper is that it introduces an I/O efficient procedure to compute the topological sort on implicitly given graphs (procedure Elim-No-Predecessors).

Our experimental evaluation confirmed that the new algorithm is able to fully solve instances of the LTL model checking problem that cannot be solved either with the standard LTL model checker SPIN or using so far the best I/O efficient approach of Edelkamp and Jabbar [14]. The approach of [14] fails especially if the verified formula is valid, which is because after the transformation, the graph becomes too large to be kept even in the external memory.

On the other hand, unlike SPIN and the approach of [14] our algorithm does not work on-the-fly. The on-the-fly algorithms are particularly successful if the property is violated and the counterexample can be found early during the state space exploration.

As our algorithm is based on the algorithm which can be easily parallelized [10], it is straightforward to develop a parallel version of the algorithm that can further speed up verification of large systems. It also seems promising to design I/O efficient variants of other BFS-based verification algorithms [3,4,8,9,10].

Some of them work on-the-fly and hence could outperform both the new algorithm and the algorithm of Edelkamp and Jabbar.

An open problem for which we still do not know a practically good solution, is the inefficiency of the delayed duplicate detection technique as used in procedure Elim-No-Predecessors. Since procedure Merge is called every time a BFS level is explored, merging a small level into a large set can slow down the exploration speed of a few vertices per minute. The question is, if this can be avoided.

## References

1. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics, Masaryk University Brno (2004)
3. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: Automated Software Engineering (ASE'03), pp. 106–115. IEEE Computer Society Press, Los Alamitos (2003)
4. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL Model-Checking in SPIN. In: Dwyer, M.B. (ed.) Model Checking Software. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
5. Barnat, J., Brim, L., Černá, I., Šimeček, P.: DiVinE – The Distributed Verification Environment. In: PDMC'05, pp. 89–94 (2005)
6. Barnat, J., Brim, L., Šimeček, P.: LTL Model Checking with I/O-Efficient Accepting Cycle Detection. Technical Report FIMU-RS-2007-01, Faculty of Informatics, Masaryk University (2007)
7. Biere, A., Artho, C., Schuppan, V.: Liveness Checking as Safety Checking. Electr. Notes Theor. Comput. Sci. 66(2) (2002)
8. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
9. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL Model Checking Based on Negative Cycle Detection. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2245, pp. 96–107. Springer, Heidelberg (2001)
10. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN'03. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
11. Chiang, Y., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., Vitter, J.: External-Memory Graph Algorithms. In: SODA'95, pp. 139–149. Society for Industrial and Applied Mathematics (1995)
12. Clarke Jr, E., Grumberg, O., Peled, D.: Model Checking. MIT press, Cambridge (1999)
13. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL Data Sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
14. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: SPIN'06, pp. 1–18 (2006)

15. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)
16. Hammer, M., Weber, M.: To Store Or Not To Store Reloaded: Reclaiming Memory On Demand. In: FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2006)
17. Holzmann, G.J., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: The SPIN Verification System, pp. 23–32. American Mathematical Society (1996)
18. Jabbar, S., Edelkamp, S.: I/O Efficient Directed Model Checking. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 313–329. Springer, Heidelberg (2005)
19. Jones, M., Mercer, E.: Explicit State Model Checking with Hopper. In: Graf, S., Mounier, L. (eds.) SPIN'04. LNCS, vol. 2989, pp. 146–150. Springer, Heidelberg (2004)
20. Katriel, I., Meyer, U.: Elementary Graph Algorithms in External Memory. In: Algorithms for Memory Hierarchies, pp. 62–84 (2002)
21. Korf, R.: Best-First Frontier Search with Delayed Duplicate Detection. In: AAAI'04, pp. 650–657. AAAI Press / The MIT Press, Cambridge, MA (2004)
22. Korf, R., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: AAAI'05, pp. 1380–1385. AAAI Press / The MIT Press, Cambridge, MA (2005)
23. Kristensen, L., Mailund, T.: Efficient Path Finding with the Sweep-Line Method Using External Storage. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
24. Kumar, V., Schwabe, E.: Improved Algorithms and Data Structures for Solving Graph Problems in External Memory. In: 8th IEEE Symposium on Parallel and Distributed Processing (SPDP'96), IEEE Computer Society Press, Los Alamitos (1996)
25. Mehlhorn, K., Meyer, U.: External-Memory Breadth-First Search with Sublinear I/O. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 723–735. Springer, Heidelberg (2002)
26. Munagala, K., Ranade, A.: I/O-Complexity of Graph Algorithms. In: SODA'99, pp. 687–694, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1999)
27. Pelánek, R.: Typical Structural Properties of State Spaces. In: Graf, S., Mounier, L. (eds.) SPIN'04. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
28. Pelánek, R.: BEEM: BEnchmarks for Explicit Model checkers (February 2007) `http://anna.fi.muni.cz/models/index.html`
29. Ravi, K., Bloem, R., Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
30. Reif, J.H.: Depth-First Search is Inherrently Sequential. Information Processing Letters 20(5), 229–234 (1985)
31. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
32. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: CAV'98, pp. 172–183 (1998)
33. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Logic in Computer Science (LICS'86), pp. 332–344. IEEE Computer Society Press, Los Alamitos (1986)
34. Vitter, J.: External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Comput. Surv. 33(2), 209–271 (2001)

# Can Flash Memory Help in Model Checking?*

Jiří Barnat[1], Luboš Brim[1], Stefan Edelkamp[2], Damian Sulewski[2], and Pavel Šimeček[1]

[1] Masaryk University Brno, Czech Republic
[2] Technische Universität Dortmund, Germany

**Abstract.** As flash media become common and their capacities and speed grow, they are becoming a practical alternative for standard mechanical drives. So far, external memory model checking algorithms have been optimized for mechanical hard disks corresponding to the model of Aggarwal and Vitter [1]. Since flash memories are essentially different, the model of Aggarwal and Vitter no longer describes their typical behavior. On such a different device, algorithms can have different complexity, which may lead to the design of completely new flash-memory-efficient algorithms. We provide a model for computation of I/O complexity on the model of Aggarwal and Vitter modified for flash memories. We discuss verification algorithms optimized for this model and compare the performance of these algorithms with approaches known from I/O efficient model checking on mechanical hard disks. We also give an answer, when the usage of flash devices pays off and whether their further evolution in speed and capacity could broaden a range, where new algorithms outperform the old ones.

## 1 Introduction

There are numerous computational tasks that require to generate and process that huge amount of data that cannot be simply kept in internal memory. Unfortunately, it is not acceptable in terms of performance to rely on the standard memory virtualization techniques provided by the operating system, and specialized algorithms must be devised to efficiently manipulate data stored externally. These are the so called I/O efficient or external-memory algorithms [2].

I/O efficient algorithms reflect physical properties of external memory devices, i.e. they are designed to minimize expensive random accesses to data in favor of their block processing. However, likewise all the PC components, also the external memory devices are being continuously developed and their properties are improving in time. Recently, flash memory based external memory devices became widely used as the so called *solid state disks* (SSDs). Unlike its magnetic counterpart, SSD does not rely on physical movements of the head(s) to access the data. Therefore, the access time is much smaller for a solid state disk

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338, the Academy of Sciences grant No. 1ET408050503, and DFG grant No. ED 74/4-1.

compared to the magnetic one. For example, the speed of random reads for a solid state disk build with NAND flash memory lies roughly at the geometric mean of the speeds of random access memory (RAM) and magnetic hard drive (HDD) [3]. The only factor limiting solid state disks from being massively spread is the cost of the device if expressed per stored bit. The cost per stored bit is still significantly higher for SSDs than for magnetic disks. However, the cost per bit is definitely subject to change in the future.

I/O efficient algorithms have been studied also in the context of formal verification, model checking [4] in particular, as one of the techniques to fight the well known state explosion problem. In this paper we focus on enumerative on-the-fly LTL model checking, which is the standard option for analyzing software systems. Our goal is to consider a simple question that comes up with the advent of solid state disks. Namely, if it is meaningful to design new I/O algorithms for LTL model checking that would take advantage of the fast random reads of a solid state disk, or if it is satisfactory to apply the existing I/O efficient LTL model checking algorithms even for SSDs whose characteristics differ significantly from the characteristics of the traditional magnetic disks.

To answer the question we design several techniques to implement an *SSD efficient* graph traversal procedure, namely we discuss several variants of hashing mechanism that is used by the Nested DFS algorithm to efficiently identify already generated states during the graph traversal. We also report on a preliminary experimental comparison of newly suggested SSD efficient and the standard I/O efficient techniques, and discuss the impact of possible technology improvements that may come in the future.

The paper is organized as follows. In Section 2 we briefly recall the standard I/O efficient techniques used for enumerative external memory LTL model-checking. In Section 3 we state the differences between the standard magnetic and new solid state disks. In Section 4 we describe several SSD efficient hashing techniques, and we show in Section 5 how these can be used to design SSD efficient Nested DFS algorithm. Section 6 report on our experimental evaluation of both the SSD and I/O efficient techniques. Finally, in Section 7 we conclude the paper and plot what impact may have possible future technological improvements.

## 2 I/O Efficient Model Checking with Mechanical Disks

LTL model checking problem can be reduced to the problem of *accepting cycle detection* in the graph [4]. In the context of enumerative LTL model checking, the graph to be searched for the presence of an accepting cycle is generated on-the-fly meaning that if a graph traversal algorithm needs to proceed to an immediate successors $t$ of a state $s$, it computes state $t$ from the state vector of $s$. To prevent re-visiting of already explored states, all states that have been processed are stored in memory, hence, if a state is generated it is first checked against the set of stored states to learn whether it is a new state or has been visited before. In the context of I/O efficient algorithms, this check is referred to as the *duplicate detection*.

Due to the huge number of states, their large size, and the speed of generating them, the memory demands while analyzing systems rise rapidly. In order to release memory, states stored in the set of visited states have to be fully or partially flushed to the external memory. Under this circumstances a check whether a state has been visited may involve I/O operation as not only the states stored in memory, but also the states stored on external memory device must be considered. This however renders a standard graph traversal algorithm inefficient as the I/O operation is in orders of magnitude slower than a single or several reads from the internal memory.

### 2.1 Graph Traversal

The core technique that gave birth to I/O efficient algorithms is the so called *delayed duplicate detection* [5,6,7,8] whose idea is to postpone the individual checks against the set of visited states and perform them together in a group amortizing thus the cost of I/O operations per a single check.

There are other techniques that have significant impact on the performance of an I/O efficient graph traversal algorithm. For example, it is possible to perform hash compaction or compression of states to be stored which results in less amount of data to be transferred between external and internal memory. Another quite successful improvement builds upon using a Bloom filter maintained in main memory in order to reduce unnecessary I/O operations. Also simple partitioning of states stored on external memory may have impact on the performance of an I/O efficient graph traversal procedure. For more details on these techniques we kindly refer the reader to [9].

As mentioned above, an important aspect of an I/O efficient algorithm is that the data stored on external memory is accessed in blocks. While the clever implementation techniques aim at reducing the number of I/O operations, or reducing the amount of data being transferred, there is also possibility to improve the performance of an I/O efficient algorithm by simple improving the performance of an I/O operation. For example, by connecting two identical external memory devices into a mirror RAID array we can achieve almost double bandwidth that the block of data may be read with from the external memory device. Note that this approach basically improves bandwidth only while does not influence the latency, i.e. the time needed to read the first bit.

Similarly, it is possible to reduce time needed for solving the problem if instead of the serial I/O efficient algorithm working over a single external device a parallel I/O efficient algorithm is used utilizing multiple external memory devices. This is, however, possible only if the algorithm involved allows parallel processing, which is, for example, the case of breadth-first search, but is not the case of depth-first search [10].

### 2.2 LTL Model Checking

For accepting cycle detection there is a space efficient optimal algorithm called *Nested Depth-First Search* [11]. Unfortunately, the algorithm becomes rather

**Table 1.** Characteristics of solid state and hard disk drives

| | HDD | SSD |
|---|---|---|
| Read Bandwidth | 65 MB/s | 72 MB/s |
| Write Bandwidth | 60 MB/s | 70 MB/s |
| Random Read Access Time | 11 ms | 0.1 ms |
| Random Write Access Time | 11 ms | 5 ms |

inefficient, as soon as states to be stored cannot be maintained in the main memory [10,12].

Recently, three different I/O efficient algorithms for solving the LTL model checking problem have been published [12,13,14]. In [12] the authors suggested to avoid the DFS-based accepting cycle detection by the reduction of the problem to the problem of testing reachability relation [15,16] whose I/O efficient solution was further improved by using the directed A* search and parallelism. Since the reduction to the reachability relation testing may result in up to quadratic increase in the space complexity, this algorithm should be rather viewed as a tool for bug hunting.

A new I/O efficient algorithm for LTL model checking was given in [13]. The algorithm avoids the expensive increase in the space complexity, but does not work on-the-fly, which means that the full state space must be generated and stored on external memory device before it is checked for the presence of an accepting cycle. This disadvantage makes the algorithm quite inefficient in the cases an error can be discovered quickly using some on-the-fly algorithm. Finally, the algorithm given in [14] is both on-the-fly and linear in the space requirements with the respect to the size of the state space.

## 3 From Mechanical to Solid State Disks

Mechanical hard disks have been around for quite a long time, and they have provided us with reliable service over these years. This is about to change with the advent of *Solid State Disks* (SSD). A solid state disk is electrically, mechanically and software compatible with a conventional (magnetic) hard disk drive. The difference is that the storage medium is not magnetic (like a hard disk) or optical (like a CD) but solid state semiconductor (NAND flash) such as battery backed RAM, EEPROM or other electrically erasable RAM-like chip. In last years, NAND flash memories outpaced DRAM in terms of bit-density [17] and the market with SSDs continues to grow. This provides faster access time than a disk, because the data can be randomly accessed and does not rely on a read/write interface head synchronising with a rotating disk. We list a typical data transfer bandwidth and access time for both magnetic and solid state disk in Table 1.

It became the standard to measure the analytical complexity of an I/O efficient algorithm using the complexity model by Aggarwal and Vitter [1]. However, for solid state disk, the model is no more valid, since it does not cover the different access times for random read and write operations. For solid state disks, we propose to extend the model of Aggarwal and Vitter with a penalty factor $p$ for random write operations.

## 4 I/O Efficient Graph Traversal with Solid State Disks

We observe that random read operations on SSDs are substantially faster than on mechanical disks, while other parameters are similar. Therefore, it appears natural to ask, whether it is necessary to employ *delayed duplicate detection* (DDD) known from the current I/O efficient graph algorithms, or it is possible to build an efficient SSD algorithm using the standard *immediate duplicate detection* (IDD), *hashing* in particular.

First, we study direct access to the solid state disk without exploiting RAM usage. This implies both random read and random write operations. The implementation serves as a reference, and can be scaled to any implicit search with a visited state space that fits on the solid state disk.

Next, we compress the state in internal memory to include the address on secondary memory only. For this case states are written sequentially to the background memory in the order of generation. For resolving hash synonyms, states lookup random reads are needed. Even though linear probing shows performance deficiencies for internal hashing, for block-wise strategies, it is the apparent candidates. Alternative hashing strategies can reduce the number of random reads.

The third option fosters flushing the internal hash table to the external device, once it becomes full. In this case, full state vectors are stored internally. For large amounts of background memory and small vector sizes, large state spaces can be looked at. Usually the exploration process is suspended while flushing the internal hash table. We observe different trade-offs for the amount of randomness for background readings and writing, which mainly depend on increasing the locality of the access.

### 4.1 Hashing

The general setting (see Fig. 1) is a background hash table $H_b$ kept on the SSD, which can hold $m = 2^b$ entries. As said, SSDs prefer sequential writes and sequential read, but can cope with an acceptable number of random reads. We additionally assume a foreground hash table $H_f$ with $m' = 2^f$ entries. The ratio between fore- and background is, therefore, $r = 2^k = 2^{b-f}$. Collisions especially on the background hash table can yield additional burden. As chaining requires overhead for storing and following links, we are left with open addressing and adequate probing strategies.

As linear probing finds elements through sequential scanning, it is I/O efficient. The efficiency analysis goes back to Knuth [18]. For a load factor of $\alpha$ a successful search requires about $1/2\,(1 + 1/(1-\alpha))$ accesses on the average, while an unsuccessful search requires about $LP_\alpha = 1/2\left(1 + 1/(1-\alpha)^2\right)$ accesses

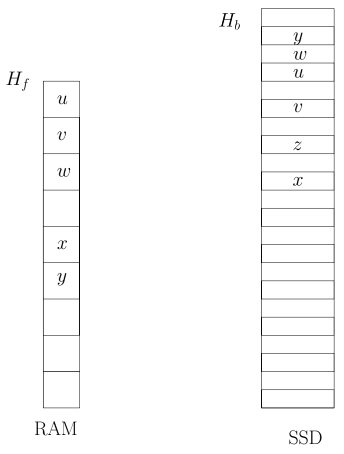


**Fig. 1.** Fore- and Background Memory, such as RAM and SSD

on the average. For a hash table that is filled up to $\alpha = 50\%$ we have less than three states to look at on the average, which easily fit into the I/O buffer. Given that random access is slower than sequential access, this implies that unless the hash table becomes filled, linear probing with one I/O per lookup per node is an appropriate option for SSD-based hashing.

## 4.2 Mapping

The simplest method to apply SSDs in graph search is to store each node at its background hash address in a file, and – if occupied – to apply conflict resolution strategy on disk. By their large seek times, this option is clearly infeasible for HDDs, but it does apply to some extent to SSDs. Nonetheless, besides extensive use of random writes that operate block-wise and are, thus, expected to be slow, one problem of the approach is the initialization time, incurred by erasing all existing data stored in background memory.

Hence, we apply a refinement to speed-up search. With one additional bit-vector array kept in RAM, we denote, whether or not a position is already occupied. This limits initialization time to reset all bits in main memory, which is much faster. Moreover, this saves lookup time in case of hashing a new state with an unused table entry. Viewed differently, one can think of a Bloom filter [19], with conflict resolution on disk. Figure 2 (left) illustrates the approach. The bit-vector *occupied* memorizes, whether the address on the SSD is in use or not.

The extra amount of RAM additionally limits the size of the search spaces to be processed. In search practice with a full state vector of several bytes to be stored in the background memory, however, investing one bit per state in RAM does not harm much, given that the ratio between main and external memory remains moderate. The only limit for the exploration is imposed by the number of states that can be stored on the solid state disk, which we assume to be sufficiently large.

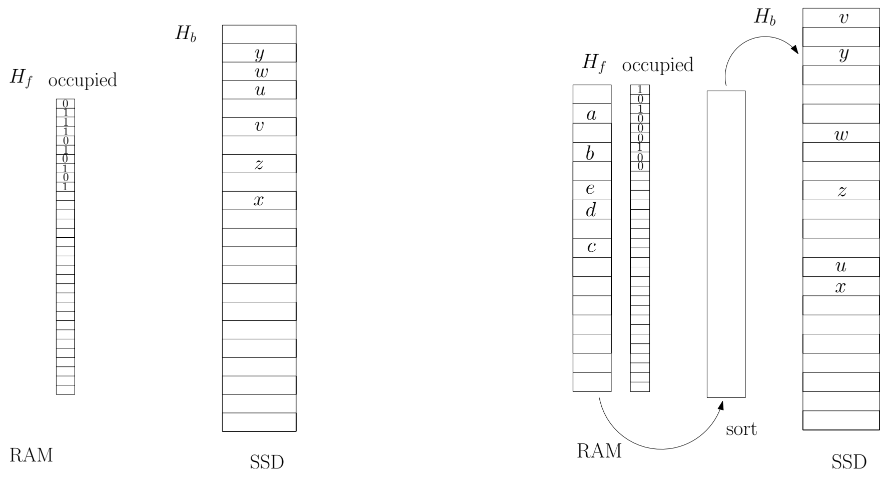


**Fig. 2.** External hashing without and with merging

For analyzing the approach, let $n$ be the number of nodes and $e$ be the number of edges in the state space graph that are looked at. Without occupied vector requires $e$ lookup and $n$ insert operations. Let $B$ is the size of a block (amount of data retrieved, or written with one I/O operation) and $|s|$ be the length of a state. As long as $LP_\alpha \cdot |s| \leq B$, at most two[1] blocks are read for each lookup[2]. For $LP_\alpha \cdot |s| > B$ no additional random read access is necessary. After the lookup, an insert operation results in one random write. This results in a flash I/O complexity of $O(e + pn)$. Using the occupied vector, the number of read operations reduces from $e$ to $n$, assuming that no collisions take place.

As the main bottleneck of the approach is random writing to the background memory, as another refinement we can additionally employ a foreground hash table as a write buffer. Due to numerous insert operations, the foreground hash table will once become filled, and then has to be flushed to the background, which incurs writes and subsequent reads. One option that we call *merging* is to sort the internal hash table wrt. to the external hash function before flushing. If the hash functions are correlated, the sequence is already presorted, by means that the number of inversions $inv(H_f) = |\{(i,j) \mid h_f(s_i) < h_f(s_j) \wedge h_b(s_i) > h_b(s_j)\}|$ is small. If $inv(H_f) = O(m')$ and given that we use an algorithm that exploits presorting[3], we obtain a linear time sorting algorithm. While flushing we now have a sequential write (due to the linear probing strategy), such that the total worst-case I/O time for flushing is bounded by the number of flushes times the efforts for sequential writes. Figure 2 (right) illustrates the approach. As we are able to exploit sequential data processing, updating the background hash table

[1] when linear probing arrives at the end of the table, an additional seek to the start of the file is needed.

[2] at our system $B = 4,096$ bytes, and $|s| \approx 40$ bytes.

[3] e.g. adaptive sort that runs in time $m' + m' \log\left(1 + \frac{inv(H_f)}{m'}\right)$.

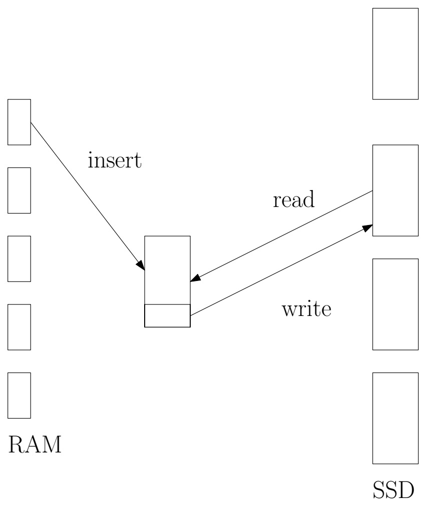


**Fig. 3.** Updating Tables in Hashing with Linear Probing while Merging

corresponds to a scan (Figure 3). Blocks are read into the RAM and merged with the internal information and then flushed back to SSD.

## 4.3 Compressing

State compression is a common option in LTL model checking. There are lossless compression strategies like FSM compaction [20], as well as lossy compression strategies like bit-state hashing [21] or hash compaction [22]. For the sake of completeness, in this paper we avoid lossy hash methods as they imply partial state space coverage.

Probably the best lossless compression ratio is obtained using practical perfect hash function [23,24]. Perfect hashing corresponds to an one-to-one mapping of some set $S$ to $\{1, \ldots, |S|\}$. Different off-line algorithms [25] have been developed that include perfect hash functions for what has been coined to the term *semi-external* LTL model checking. We do not apply perfect hashing at all, as for the construction of perfect hash functions, set $S$ has to be known, which contradicts the purpose of on-the-fly model checking.

Instead we store all state vectors in a file on the external storage device, and substitute the state vector by its relative file pointer position. For an external hash table of size $m$ this requires $\lceil \log m \rceil$ bits per entry, that is $m\lceil \log m \rceil$ bits in total. Figure 4 illustrates the approach with arrows denoting the position on external memory. An additional bit-vector *occupied* is no longer needed.

This strategy also results in $e$ lookups and $n$ insert operations. Since the ordering of states on the SSD does not necessarily correlate with the order in main memory, the lookup of states due to linear probing induces multiple random reads. Hence, the amount of individual blocks which have to be read is bounded by $LP_\alpha \cdot e$. In contrast, all insert operations are performed sequentially, utilizing a cache of $B$ bytes in memory. Subsequently this approach performs $O(LP_\alpha \cdot e)$

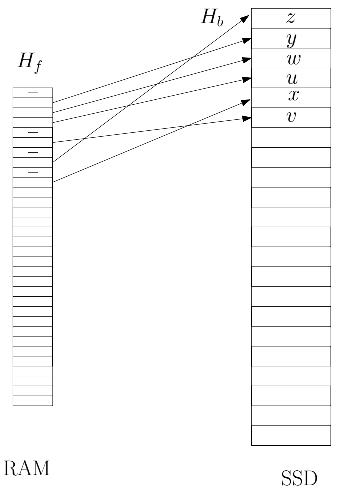


**Fig. 4.** State Compressing

random reads to the SSD. As long as $LP_\alpha < 2$ this approach performs less random read operations then mapping. By using another internal hashing strategy, e.g. cuckoo hashing [26] one reduces the number of lookups to at most 2. As sequential writing of $n$ states of $s$ bytes requires $n|s|/B$ I/Os, the total flash-memory I/O complexity is $O(LP_\alpha \cdot e + n|s|/B)$.

## 4.4 Flushing

The above approaches either require significant time to write data according to $h_b$, or request significant sizes of foreground memory. There are further trade-offs that we will consider next.

One first solution that we call *padding* is to append the entire foreground hash table as it is to the existing data on the background table. Hence, the background hash function can be roughly characterized as $h_b(s) = i \cdot m' + h_f(s)$, where $i$ denotes the current number of flushes, and $s$ the state to be hashed.

Writing is sequential, and conflict resolution strategy is inherited from the internal memory. For several flushing reading a state for answering membership queries becomes involved, as the search for one state incurs up to $r$ many table lookups. Conflict resolution may lead to an even worse performance. For a moderate number of states that exceed RAM resources only by a very small factor, however, the average performance is expected to be good. As far as all states can reside in main memory no access to the background memory is needed.

We can safely assume that load factor $\alpha$ is small enough, so that the extra amount of work due to linear probing is transparent by using block accesses. Again $e$ lookups and $n$ insert operations are performed. Let $e_i$ be the number of successors generated in stage $i$, $i \in \{0, \dots, r-1\}$. For stage 0 no access to the background table is needed. For stage $i$, $i > 0$, at most $O(i \cdot e_i)$ blocks have to be read. Together with the sequential write of $n$ elements (in $r$ rounds) this results in a flash memory complexity of $O(n|s|/B + rp + \sum_{0 \le i < r} i \cdot e_i)$ I/Os.

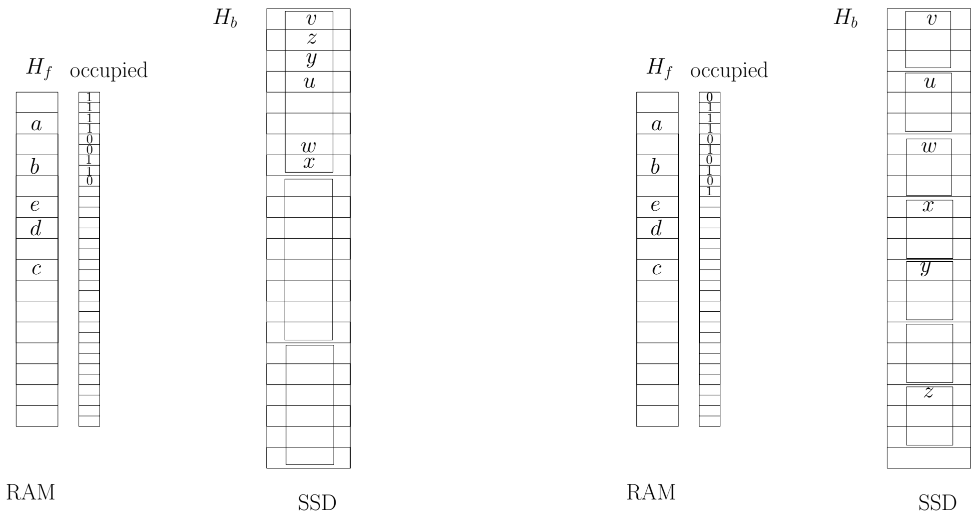


**Fig. 5.** Padding and slicing

An illustration is provided in Figure 5 (left). The entire foreground hash table has been flushed once, while the maximum number of flushes is set to 3.

The obvious alternative is to *slice* the background hash table such that $h_b(s)$ becomes $h_f(s) \cdot r + i$. An illustration is provided in Figure 5 (right); situation after one flush, and, again, at most 3 flushes are assumed.

The disadvantage of processing the entire external hash table during flushing is compensated by the fact that the probing sequences in the hash tables can now be searched concurrently. For the lookup we use a Boolean vector of size $i$ that monitors if an individual probing sequence has terminated with an empty bucket. If all probing sequences fail, the query itself has failed.

## 5 I/O Efficient Model Checking with Solid State Disks

In Section 4 various implementations of graph traversal with SSD are shown. It is apparent that some of them are less I/O efficient, but have lower demands on the internal memory (mapping and flushing strategies), while others allocate more of RAM, but perform much less I/O operations in the ordinary case (compress strategy).

On the basis of these graph traversals, it is relatively easy to construct LTL model checking algorithms. Nested DFS, as introduced above, can be implemented with two independent hash tables. To save space it is, however, recommended to use one hash table for storing the states and one internal bit-vector array *flagged* to memorize if a state has been visited in the second depth-first search.

With the above hashing schemes, we arrive at full flexibility in applying immediate duplicate detection in Nested DFS. Table 2 summarizes the hash functions applied and the amount of memory required for the different hashing strategies in

**Table 2.** Trademarks for different hash strategies for on-the-fly LTL model checking algorithm. Upper two lines give an overview of hash functions, lower three lines show a space complexity in bits for different levels in memory hierarchy.

| | Mapping | Compressing | Padding | Slicing |
|---|---|---|---|---|
| $h_f$ | – | $h_d mod\ m$ | $h_d mod\ m$ | $h_d mod\ m$ |
| $h_b$ | $h_d mod\ m$ | $h_d mod\ m$ | $i \cdot m' + h_d mod\ m'$ | $(h_d mod\ m) \cdot r + i$ |
| RAM | $2m$ | $m + m\lceil \log m \rceil$ | $2m + m' \times \lvert s \rvert$ | $2m + m' \times \lvert s \rvert$ |
| SSD | $m \times \lvert s \rvert$ | $m \times \lvert s \rvert$ | $m \times \lvert s \rvert$ | $m \times \lvert s \rvert$ |
| HDD | $\max_i \lvert Open_i \rvert \times \lvert s \rvert$ | $\max_i \lvert Open_i \rvert \times \lvert s \rvert$ | $\max_i \lvert Open_i \rvert \times \lvert s \rvert$ | $\max_i \lvert Open_i \rvert \times \lvert s \rvert$ |

$m = 2^b$, $m' = 2^f$, $h_d$ is hash function in DiVinE [27], $m$ is the size of background hash table (in the number of elements), $|s|$ is state vector size (measured in bits), $Open_i$ is the number of states in the search frontier in iteration $i$.

LTL model checking. Note that there are recent refinements to Nested DFS [28] that are faster, but need more bits.

## 6 Experimental Evaluation

We implemented our algorithms in DiVinE (DIstributed VerIficatioN Environment) [27], including only part of the library deployed with DiVinE, namely state generation and internal storage. For the implementation of external-memory container and for algorithms for efficient sorting and scanning we use STXXL (Standard Template Library for Extra Large Data Sets) [29]. Models are taken from the BEEM library [30].

For the first set of experiments we used a Desktop PC with AMD Athlon CPU (32 bit) a SATA HDD of 280 GB with 13.8 ms seek time and about 61.5 MB/s for sequential reading and a 32 GB 3.5" SATA high-speed flash memory solid state disk (HAMA), which has 0.14 ms seek time and scales to about 93 MB/s for sequential reading.

To confirm the theoretical results we check the Rether-4 protocol from the BEEM library (Fig. 6). The plot shows Nested DFS runs with different immediate duplicate detection strategies. All experiments, aside from the *mapping* strategy, were stopped after 40,000s (this strategy was stopped after 1,800s due to its obvious lack of performance). The mapping strategy is the worst one because of numerous random writes. We use padding as a flushing strategy. As linear probing is used to store the positions of the saved states, we observe an increased number of *read* operations as the internal hash table becomes filled. Compress strategy appears to perform the best, which corresponds to its I/O complexity without any penalties for random write operations. The difference between *compress* and *compress (stack on hdd)* is the location of the stack file. In the first case, it was located on the SSD, in the second it was on a separate HDD. We observe that having the stacks stored on a second hard disk gives another speed-up of about 30% for the state space traversal.

The motivation for use of SSDs was to exploit fast random access to them. Now, we compare new algorithms designed for SSDs to traditional I/O efficient

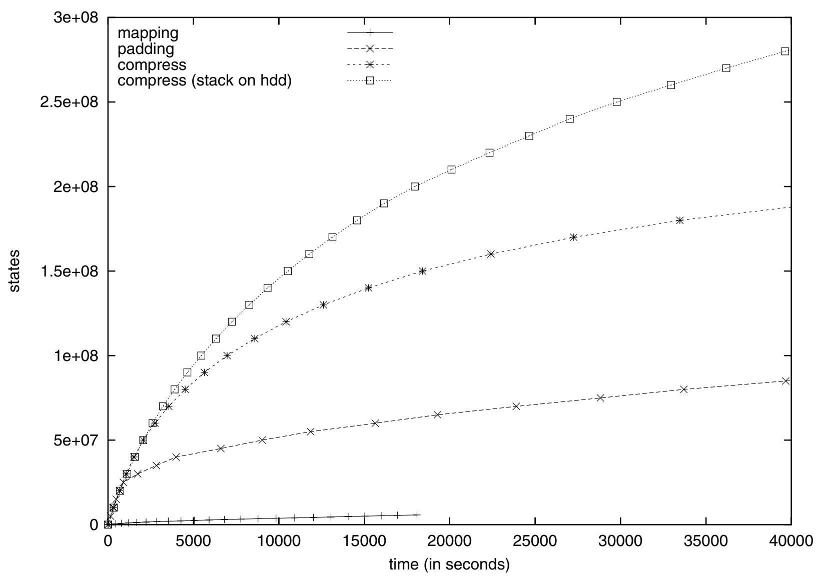


**Fig. 6.** Comparing the three strategies on the Rether-4 model

algorithms, which we run on SSDs too. To get a fair picture about both approaches, we perform a reachability analysis in breadth first order. As a novel approach we run BFS with immediate duplicate detection and compression strategy (Compress BFS). As a traditional approach we run a standard external BFS with delayed duplicate detection after each level (External BFS).

First, the state space of the Szymanski (5 prop4) model was generated using both approaches. The plot in Fig. 7 demonstrates the dependency in expanding speed between the Compress BFS and the BFS layer size, while the expanding time per layer remains almost the same for External BFS. This is due to the fact, that in delayed duplicate detection the time of level generation is mostly determined by the size of the visited states set, which is completely passed for each BFS layer. Thus, in large search depth, immediate duplicate detection saves much time, compared to delayed duplicate detection.

Therefore, it is apparent that results strongly depend on a structure of a state space. Provided that I/O complexity of External BFS is $O((e/m + \#layers)(n|s|/B)$ [13], it is clear that its I/O complexity is highly dependent on the number of BFS layers, while the I/O complexity of Compress BFS is not. This can be demonstrated on the model Rether-2, with 552 BFS layers (see Fig. 8). While External BFS performs poor on this model, Compress BFS finishes in several minutes. The new approach can also benefit from a small number of back edges and various heuristics helping to recognize duplicates with no reading from disk. This is a case of model Train-Gate, where the amount of random reads was only 30 million, even though the state space has 50 million states, due to the fact that duplicates were typically found in internal buffers (only 8 MB large) before flushing to disk. Model MCS is an example, where External BFS performs better – the state space has relatively low number of BFS levels (90).

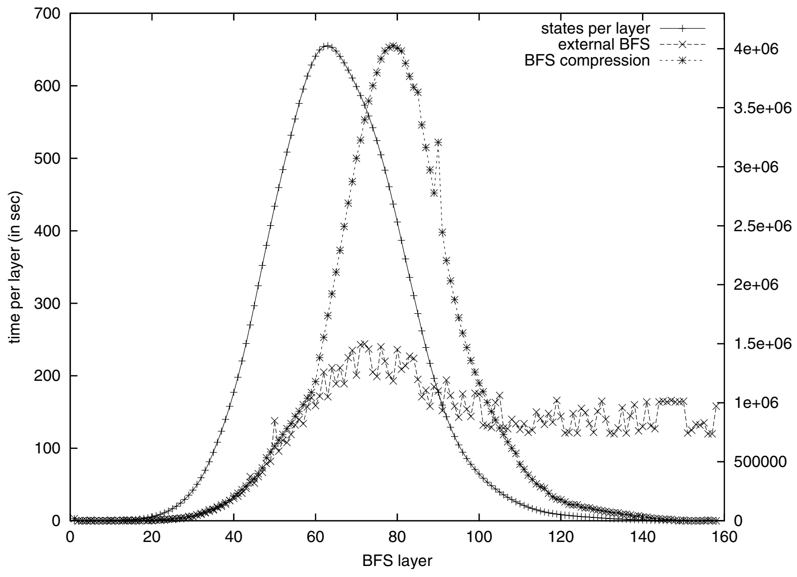
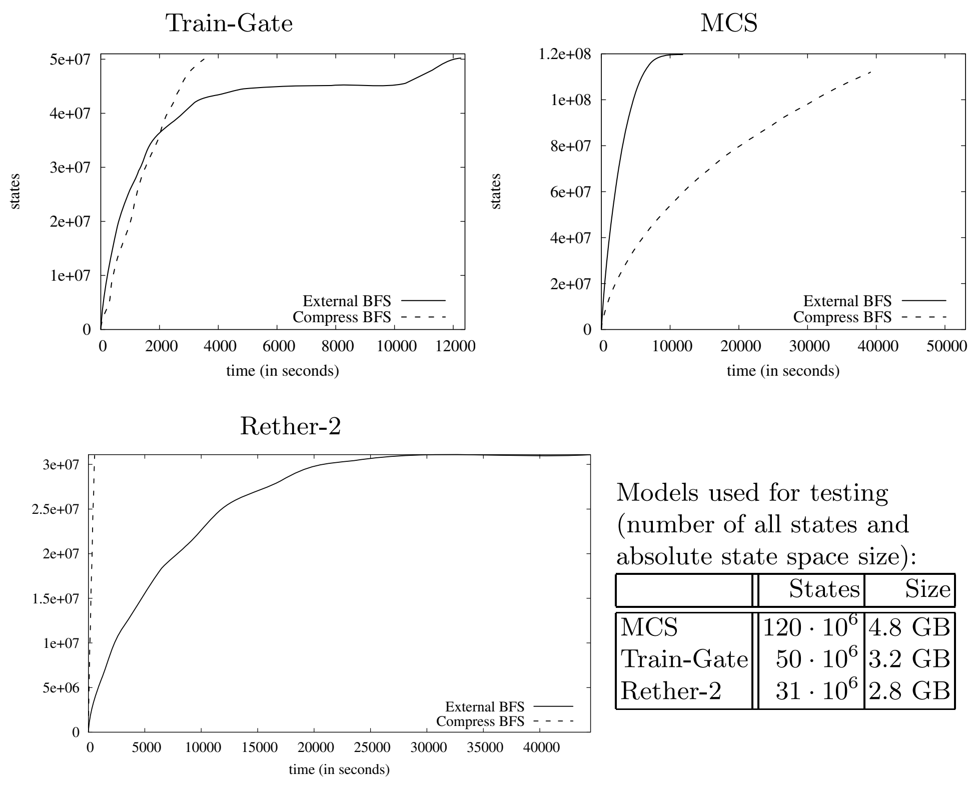


**Fig. 7.** Comparing Compress BFS to External BFS on the Szymanski 5 prop4 model. The right axis, together with the crossed plot shows the size of each layer. The remaining curves shows time per layer for the different approaches.



Models used for testing (number of all states and absolute state space size):

| | States | Size |
|---|---|---|
| MCS | $120 \cdot 10^6$ | 4.8 GB |
| Train-Gate | $50 \cdot 10^6$ | 3.2 GB |
| Rether-2 | $31 \cdot 10^6$ | 2.8 GB |

**Fig. 8.** Comparison of External BFS and Compress BFS

From the I/O complexities of both algorithms and from our measurements it follows that External BFS has to slow down the exploration faster than Compress BFS with increasing portion of the state space explored. Thus, Compress BFS can often outperform it from some BFS level due to its linearity in I/O complexity. The moment, when Compress BFS outperforms External BFS depends to high extent on numerous platform and input specific factors: state space structure (number of BFS layers, portion of back edges), bandwidth, access time, file system, implementation (we did not implemented heuristics from [14] or [9]). Even though it is not easy to predict, whether or from which point of exploration Compress BFS outperforms External BFS, the main impact of behaviour of both algorithms is that there can be a threshold, from which Compress BFS outperforms External BFS on a given input and so algorithms for SSDs like Compress BFS are practical.

## 7 Conclusions

We have contributed several new approaches to hashing applied to SSDs. The most important observation is with the advent of SSD technology, immediate duplicate detection becomes tractable, offering much more flexibility for the choice of the exploration strategy. Monitoring CPU performance, we observed hashing strategies preserve ratios of 50% or more, suggesting that I/O waits are present, but not thrashing. With SSDs random access time decreasing, SSDs will likely become fast enough to rise the CPU usage to 100% making the SSD fully transparent to the user[4].

Compression, the best performing strategy, requires substantial main memory, which according to current ratios of space between RAM and SSDs is still no bottleneck. Although we have tested DFS and BFS, non heuristic algorithms, our SSD hashing strategies can also be applied to heuristic approaches, e.g. A* to rise the amount of states that can be visited. Using SSDs as a shared external storage device for cluster computers will result in an even higher throughput, even for random reads, giving a better possibility for parallel processing.

Directly compared to standard I/O algorithms, for a given model there can be a threshold in state space exploration, from which these new approaches pay off due to their linearity in size of state space – at least for the compress approach. Traditional I/O efficient algorithms are not linear, but they have good constant factors which allow them to outperform new approaches on many inputs. If the bandwidth of SSDs will grow faster, traditional I/O algorithms pay off. If the access time of SSDs will decrease faster than their bandwidth, the importance of new approaches will increase.

Due to easiness of parallel disk connection, large capacities of SSD are possible[5] . Nevertheless, prices for SDDs are still high. Fortunately, in last years they decrease reasonably as the market with flash memories grows.

[4] According to Dell, current prices for 32GB RAM are 6 times higher than for 32GB SSDs.

[5] E.g. StorgeSpire – 1 TB SDD array by Solid Data (`http://www.soliddata.com/products/storagespire`).

## References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Sanders, P., Meyer, U., Sibeyn, J.F.: Algorithms for Memory Hierarchies. Springer, Heidelberg (2002)
3. Min, S.L., Nam, E.H., Lee, Y.H.: Evolution of NAND flash memory interface. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697, pp. 75–79. Springer, Heidelberg (2007)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. Korf, R.: Best-First Frontier Search with Delayed Duplicate Detection. In: AAAI 2004, pp. 650–657. AAAI Press / The MIT Press (2004)
6. Korf, R., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: AAAI 2005, pp. 1380–1385. AAAI Press / The MIT Press (2005)
7. Munagala, K., Ranade, A.: I/O-Complexity of Graph Algorithms. In: SODA 1999, Philadelphia, PA, USA, pp. 687–694. Society for Industrial and Applied Mathematics (1999)
8. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
9. Hammer, M., Weber, M.: To Store Or Not To Store Reloaded: Reclaiming Memory On Demand. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
10. Barnat, J.: Distributed Memory LTL Model Checking. PhD thesis, Faculty of Informatics, Masaryk University Brno (2004)
11. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. 1(2-3), 275–288 (1992)
12. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
13. Barnat, J., Brim, L., Šimeček, P.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
14. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
15. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. 66(2) (2002)
16. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
17. Kim, K., Choi, J.H., Choi, J., Jeong, H.S.: The future prospect of nonvolatile memory. In: 2005 IEEE VLSI-TSA International Symposium on VLSI Technology (VLSI-TSA-Tech), pp. 88–94 (2005)
18. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison Wesley, Reading (1973)
19. Bloom, B.: Space/time trade-offs in hashing coding with allowable errors. Communication of the ACM 13(7), 422–426 (1970)

20. Holzmann, G.J., Puri, A.: A minimized automaton representation of reachable states. International Journal on Software Tools for Technology Transfer 2(3), 270–278 (1999)
21. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods in System Design 13(3), 287–305 (1998)
22. Stern, U., Dill, D.L.: Combining state space caching and hash compaction. In: Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop, pp. 81–90. Shaker Verlag, Aachen (1996)
23. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 139–150. Springer, Heidelberg (2007)
24. Botelho, F.C., Ziviani, N.: External perfect hashing for very large key sets. In: CIKM 2007: Proceedings of the sixteenth ACM Conference on information and knowledge management, pp. 653–662 (2007)
25. Edelkamp, S., Sanders, P., Simecek, P.: Semi-external LTL model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 530–542. Springer, Heidelberg (2008)
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
27. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
28. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
29. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
30. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)

# Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking*

J. Barnat[1], L. Brim[1], P. Šimeček[1], and M. Weber[2]

[1] Masaryk University Brno, Czech Republic
[2] University of Twente, The Netherlands

**Abstract.** Revisiting resistant graph algorithms are those, whose correctness is not vulnerable to repeated edge exploration. Revisiting resistant I/O efficient graph algorithms exhibit considerable speed-up in practice in comparison to non-revisiting resistant algorithms. In the paper we present a new revisiting resistant I/O efficient LTL model checking algorithm. We analyze its theoretical I/O complexity and we experimentally compare its performance to already existing I/O efficient LTL model checking algorithms.

## 1 Introduction

Model checking real-life industrial systems is a memory demanding and computation intensive task. Utilizing the increase of computational resources available for the verification process is indispensable to handle these complex systems. The three major approaches to gain more computational power include the usage of parallel computers, clusters of workstations and the usage of external memory devices (hard disks), as well as their combination.

In this paper, we focus on external memory devices, where the goal is to develop algorithms that reduce the number of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on an external device is orders of magnitude slower than the access to information stored in main memory. Thus, the complexity of I/O efficient algorithms is measured in the number of I/O operations [1].

The automata-theoretic approach [2] to model checking finite-state systems against linear-time temporal logic (LTL) reduces to the detection of reachable accepting cycles in a directed graph. Recently, two I/O efficient LTL model-checking algorithms that allow verification of both safety and liveness properties have been proposed in [3] and in [4]. Both algorithms build on breadth-first traversal through the graph and employ the *delayed duplicate detection* technique [5,6,7,8]. The traversal procedure has to maintain a set of visited vertices (*closed set*) to prevent their re-exploration. Since the graphs are large, the closed set cannot be kept completely in main memory. Most of it is stored on an external memory device. When a new vertex is generated (into the *open set*) it is checked against the closed set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

together in a group, for the price of a single scan operation. We assume that the delayed vertices are stored in the main memory as *candidate set*. In order to minimize the number of scan operations which merge the closed set on disk with the candidate set, it is important that the candidate set is as large as possible. In the case of BFS traversal, candidate sets are formed typically from a single BFS level. However, if the level is small, the utility of delaying the duplicate check drops down. A possible solution is to maximize the size of the candidate set by exploring more BFS levels at once. This, in general, leads to revisiting of vertices due to cycles and might violate the correctness of the algorithm. Whether correctness is preserved depends on the algorithm itself. E.g., if an algorithm uses BFS to traverse the reachable part of a graph, revisiting of vertices does not disturb its correctness, while the algorithm for computing a topological sort is not resistant to such revisits.

It is important to note that even though vertex revisits result in performing more (cheap) RAM operations, it might significantly reduce the number of expensive I/O operations. Thus, *revisiting resistant* algorithms are expected to be more I/O efficient than non-resistant ones in practice. In the first part of the paper we explore the notion of a revisiting resistant graph algorithm in more detail.

We are interested in LTL model-checking algorithms for very large *implicit graphs*, i.e., graphs defined by an initial vertex and a successor function. In previous work, we provided an I/O efficient LTL model checking algorithm that builds on topological sort [4]. The algorithm does not work on-the-fly, however, which limits its applicability. In addition, the algorithm is not revisiting resistant. The main contribution of this paper is to overcome these obstacles by providing a new algorithm. The algorithm adapts the idea of the on-the-fly MAP algorithm [9], which is revisiting resistant. In particular, we exploit the algorithm's property of decomposing a graph into several independently processable, smaller sub-graphs. This, in combination with revisiting resistance, significantly improves its practical behavior. We consider several heuristics that guide the decomposition.

*Related work.* Regarding I/O efficient LTL model-checking, we explicitly compare our work to all existing approaches in Sections 5 and 6. Works on improving the efficiency of delayed duplicate detection (DDD) include hash-based DDD [10], structured DDD [11], graph compression, lossy hash tables and bit-state hashing [12]. All these techniques are orthogonal to our approach and can be combined with the revisiting resistance principle. We have not implemented these other techniques to provide an empirical evaluation.

*Main Results.* The contribution of this paper can be summarized as follows:

– We explore the notion of a revisiting resistant algorithm and show that the I/O efficient algorithm from [4] is not revisiting resistant (Section 2).
– We present a revisiting resistant I/O efficient reachability algorithm (Section 3).
– We describe the I/O efficient MAP algorithm for LTL model-checking that works *on-the-fly* (Section 4), analyze its theoretical complexity (Section 5), and compare it to other algorithms, both in terms of asymptotic complexity (Section 5) and experimental behavior (Section 6).

## 2 Revisiting Resistance

In this section, we explain that some algorithms exhibit a quite distinct property that can be of use when adapting the algorithm to an I/O efficient setting. We will refer to this property as *revisiting resistance* and will brand algorithms satisfying the property as *revisiting resistant algorithms*.

We start with a brief description of a general graph search algorithm. Basically, a search algorithm maintains two data structures: a set of vertices that await processing (*open set*) and a set of vertices that have been processed already (*closed set*). The way in which vertices are manipulated by a general algorithm is depicted in Fig. 1(a). A vertex from the open set is selected and its immediate successors are generated (by traversing edges originating from the vertex). The newly generated vertices are checked against the closed set, to ensure that information stored in the closed set is properly updated. Also, if there is need for further processing of some vertices, they are inserted into the open set along with all necessary information for the processing.

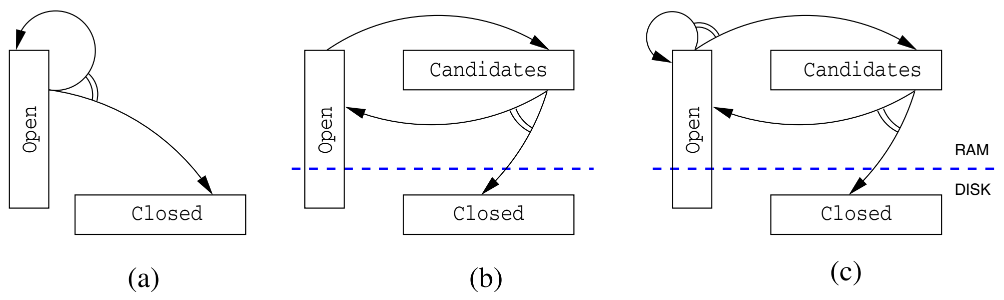
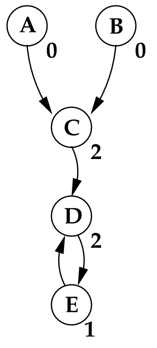


**Fig. 1.** Vertex work flow: (a) Standard search algorithm, (b) I/O search algorithm with delayed duplicate detection, (c) I/O search algorithm with delayed duplicate detection and revisiting

An I/O efficient search algorithm utilizing the delayed duplicate detection technique has a different vertex work flow. A vertex is picked from the open set and its successors are generated. Then they are inserted into the set of *candidates*, i. e., vertices for which the corresponding check against the closed set has been postponed. In our approach, the set of candidates is kept completely in memory. Candidates are flushed to disk using a *merge* operation under two different circumstances: Either the open set runs empty and the algorithm has to perform a merge to get new vertices into it, or the candidate set is too large and cannot be kept in memory anymore. The merge operation performs the duplicate check of candidate vertices against closed vertices, and inserts those vertices that require further processing into the open set. A schema of the vertex work flow is depicted in Fig. 1(b).

As explained, the merge operation is performed every time the algorithm empties the set of open vertices. Under the standard I/O efficient approach to BFS graph exploration this happens at least after every BFS level. We have observed that for many particular runs of the I/O efficient BFS algorithm, the fact that the merge operation appears after every BFS level is actually a weak point in the practical performance of the algorithm. This is because often a single BFS level contains a relatively small number of vertices,



| Open set | Delayed updates to the closed set | Impact of merge operation on the closed set |
|---|---|---|
| $A,B$ | $\emptyset$ | $(A,0),(B,0),(C,2),(D,2),(E,1)$ |
| $B,C$ | $(C,-1)$ due to $A \to C$ | $(A,0),(B,0),(C,1),(D,2),(E,1)$ |
| $C,C$ | $(C,-1)$ due to $B \to C$ | $(A,0),(B,0),(C,0),(D,2),(E,1)$ |
| $C,D$ | $(D,-1)$ due to $C \to D$ | $(A,0),(B,0),(C,0),(D,1),(E,1)$ |
| $D,D$ | $(D,-1)$ due to $C \to D$ | $(A,0),(B,0),(C,0),\boxed{\mathbf{(D,0)}},(E,1)$ |

**Fig. 2.** Example computation of the topological-sort based cycle detection algorithm. Values associated with a vertex correspond to the number of immediate predecessors that have not been processed yet. After the computation, vertices that are associated with a zero value lie outside cycles. The algorithm is not revisiting resistant, as vertex $D$ is labeled with a zero value after the merge operation, although it does lie on a cycle.

in comparison to the full graph. Processing them means that the merge operation has to traverse a large disk file, which is costly.

To fight this inefficiency, we suggest a modification in the vertex work flow of an I/O efficient algorithm, as depicted in Fig. 1(c). A vertex, when generated, is inserted not only into the set of candidates, but also into the open set. This causes some of the vertices stored in the candidate set to be revisited. I.e., the "visit" procedure is performed repeatedly for a vertex *without* properly updating its associated information in the closed set residing in external memory. Consequently, some graph algorithms may exhibit incorrect behavior. Revisiting resistant external memory graph algorithms are those, whose correctness is not vulnerable to repeated edge exploration from vertices in the open set. Below, we demonstrate that some algorithms are vulnerable to the revisiting of candidates and become incorrect, while others cope with the revisiting without problems.

We exemplify the concept of revisiting resistance on the *single source shortest path* (SSSP) and the *topological-search based cycle detection* (OWCTY) [13,14,15] algorithms. As for the SSSP algorithm, the procedure that is bound to edge exploration computes a new value for the target vertex, or updates it if the newly computed value is better (lower in this case) than the value stored before. For example: suppose, an edge $(u,v)$ is labeled with the value $t$, and vertex $u$ is stored in the open set with an associated value $d(u)$, representing the length of the currently shortest known path to vertex $u$ from the source vertex. The procedure computes a new value for vertex $v$ using the formula $d(v) = d(u) + t$. The new pair $(v, d(v))$ is stored in the candidate set and awaits merging with the value stored in the closed set. After the merge, the value stored in the closed set corresponds to the minimum of the originally stored value and the newly computed value. Note, that the resulting value in the closed set is independent from the number of re-explorations of edge $(u,v)$, and, in other words, the number of merges. Even if performed several times, the computation of the minimum among several possible values remains correct. Therefore, we consider the SSSP algorithm as revisiting resistant.

The situation is quite different in the case of the I/O-efficient OWCTY algorithm. The algorithm performs a cycle detection that is based on the recursive elimination of vertices with zero predecessors. At first, the algorithm computes the number of immediate predecessors for every reachable vertex, and then eliminates vertices whose predecessor count drops to zero. During vertex elimination, the predecessor count is decreased for all immediate successors of the eliminated vertex. Thus, when visiting a vertex $v$ from vertex $u$, the predecessor count stored at vertex $v$ has to be decreased by one. Unfortunately, the value stored for vertex $v$ can be maintained correctly only by costly access to external memory. One way around this dilemma is to only store a delta alongside $v$ in the candidate set. E. g., a pair $(v,-1)$ indicates that there is a new eliminated predecessor of $v$, and so the value associated with $v$ should be decreased by one. If we now allow the algorithm to further explore vertices below $v$, it may happen that the edge $(u,v)$ is re-explored again and another pair $(v,-1)$ is inserted into the set of candidates. However, when the set of candidates is merged with the closed set on disk, the predecessor count of vertex $v$ gets decreased *twice*. This violates the correctness of the algorithm. The problem is exemplified in Fig. 2. Therefore, the algorithm is not revisiting resistant.

## 3 Revisiting Resistant Reachability

This section explains a simple revisiting resistant algorithm, an I/O efficient breadth-first search (Alg. REACHABILITY). The algorithm's sole task is to traverse all vertices, thus we only have to remember for each vertex whether it has been visited or not. Clearly, once a vertex is marked as visited, additional visits do not change this property.

We introduce revisiting resistance to the standard I/O efficient breadth-first search (BFS) procedure as follows. After a single BFS level is fully generated, a decision is made whether the set of candidates will be merged with the closed set, or whether another BFS level will be processed without prior merging. The pseudo-code of function OPENISNOTEMPTY makes this more precise.

According to our observations, omitting the merge operation as long as there is still some unused memory left is not the optimal strategy. Merging with a small closed set might be cheaper than repeatedly re-exploring vertices of the candidate set. We avoid postponing merge operations on small closed sets by introducing a decision formula that builds upon the estimated time needed to fully generate the next BFS level $(n+1)$, and the estimated time needed to perform the merge operation. These estimations are denoted as $estim(t^{gen}_{n+1})$ and $estim(t^{merge}_{n+1})$, respectively, and they are computed from the sizes of open and closed sets as follows:

$$estim(t^{gen}_{n+1}) = t^{gen}_{n} \cdot \frac{|Open_{n+1}|}{|Open_n|} \quad \text{and} \quad estim(t^{merge}_{n+1}) = t^{merge}_{n} \cdot \frac{|Closed_n| + |Open_n|}{|Closed_n|},$$

where $|Open_{n+1}|$ refers to the number of newly discovered vertices w.r.t. *Candidates.* The decision formula is then a simple comparison of the estimated values:

$$estim(t^{gen}_{n+1}) < estim(t^{merge}_{n+1})$$

Finally, note that in our approach the entire set of candidates is kept in memory. However, there is a different approach to I/O efficient reachability, in which both, the

**Algorithm 1.** REACHABILITY

```
1: $OmittedMergeCount \leftarrow 0$
2: $Candidates \leftarrow \emptyset$
3: $s \leftarrow$ GETINITIALVERTEX()
4: $Closed \leftarrow \{s\}$
5: $Open.push(s)$
6: while OPENISNOTEMPTY() do
7:     $s \leftarrow Open.pop()$
8:     for all $t \in$ GETSUCCESSORS($s$) do
9:         if $t \notin Candidates$ then
10:            $Candidates \leftarrow Candidates \cup \{t\}$
11:            $LastLevel \leftarrow LastLevel \cup t$
12:            if MEMORYISFULL() then
13:                MERGE()
```

**Algorithm 2.** OPENISNOTEMPTY

```
1: if $Open.isEmpty()$ then
2:     if ESTIMGEN() < ESTIMMERGE() then
3:         $Open.swap(LastLevel)$
4:         $OmittedMergeCount \leftarrow OmittedMergeCount + 1$
5:     else
6:         MERGE()
7: return $\neg Open.isEmpty()$
```

**Algorithm 3.** MERGE

```
1: for all $s \in Closed$ do
2:     if $s \in Candidates$ then
3:         $Candidates \leftarrow Candidates \setminus \{s\}$
4: if $OmittedMergeCount > 0$ then
5:     $Open \leftarrow (Open \cup LastLevel) \setminus Closed$
6:     $OmittedMergeCount \leftarrow 0$
7: for all $s \in Candidates$ do
8:     if $OmittedMergeCount = 0$ then
9:         $Open.push(s)$
10:    $Closed \leftarrow Closed \cup \{s\}$
11: $LastLevel.clear()$
12: $Candidates \leftarrow \emptyset$
```

candidate and the closed set are stored in external memory [3]. It is possible to combine revisiting resistant algorithms with that approach by employing a similar decision formula to trigger the merge operation.

## 4 I/O Efficient MAP Algorithm

In this section we design a new revisiting resistant, I/O efficient algorithm for detecting reachable accepting cycles in an implicitly given, directed graph. The algorithm is derived from the *Maximal Accepting Predecessors* (MAP) algorithm [9,16]. We discuss advantages and disadvantages of the algorithm in comparison to other I/O efficient LTL model checking algorithms.

The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own accepting predecessor. Instead of expensive computing and storing of all accepting predecessors for each (accepting) vertex, the algorithm computes and stores a single representative accepting predecessor for each vertex, namely the maximal one in a suitable ordering of vertices.

Let $G = (V, E, s_0, F)$ be a directed graph, where $V$ is a set of vertices, $E$ is a set of edges, $s_0$ is an initial vertex, and $F$ a set of accepting vertices. For technical reasons we assume $s_0$ is not accepting. Let $\prec$ be a linear order on vertices with minimal element $\perp$. Let $u \rightsquigarrow^+ v$ denote that there is a directed path from $u$ to $v$. Then the *maximal accepting predecessor* function $map_G$ is defined as:

$$map_G(v) = \max\left(\{u \in F \mid u \rightsquigarrow^+ v\} \cup \{\perp\}\right)$$

Accepting cycle detection is based on the fact that if $v = map_G(v)$, then $v$ lies on an accepting cycle. While the condition is sufficient, it is not necessary—it is possible that $v \neq map_G(v)$ but $v$ lies on an accepting cycle. Moreover, if $v$ is accepting and $map_G(v) \prec v$ then $v$ does not lie on an accepting cycle. Therefore, the MAP algorithm repeatedly removes vertices for which $map_G(v) \prec v$ and recomputes $map_G$ for all vertices.

There is another feature of the MAP algorithm that can be exploited in designing its I/O efficient version. If $map_G(u) \neq map_G(v)$, then $u$ and $v$ cannot lie on the same accepting cycle. This property allows to decompose the graph $G$ into disjoint subgraphs each time $map_G$ values are computed. Let $P(u) = \{v \mid map_G(v) = u\}$. The vertex $u$ is called a *seed* of a *partition* $P(u)$. For any two vertices $u, v$ we have that either $P(u) = P(v)$ or $P(u) \cap P(v) = \emptyset$. The subgraphs are given by disjoint partitions and it is enough to search for accepting cycles in each partition separately. The algorithm thus maintains a queue of pairs $(seed, partition)$, which is initialized with the partition containing all vertices of $G$ and the initial vertex $s_0$ as its seed. On each partition $P$ the algorithm computes $map_G$ values. If there is a vertex $u$ such that $map_G(u) \in P$, an accepting cycle is detected, otherwise $P$ is split into smaller partitions which are stored for further processing. Note that sub-partitions with $map_G = \perp$ are dropped immediately.

The algorithm obtains the necessary linear ordering on vertices by assigning a unique number to every vertex. This also allows us to store for each vertex the *order value* of its maximal accepting predecessor rather than the maximal accepting predecessor itself.

The basic structure of our new algorithm follows the original MAP algorithm. We maintain a queue *Partitions* of unprocessed partitions as produced by computing $map_G$. For each partition the algorithm also records its size. If a partition fits into the main memory, we call a standard in-memory algorithm searching accepting cycles, for example, *nested depth-first search* [17].

Procedure MAP propagates the highest order values across the graph in order to compute $map_G$ values for all vertices in a given partition. In essence, the procedure is very similar to external BFS, but it allows to enqueue already explored vertices if it increases their $map_G$ value. We return to this point in the explanation of procedure UPDATEMAP.

**Algorithm 4.** DETECTACCEPTINGCYCLE$(G)$

```
Require: $G = (V, E, s_0, F)$
1: $Partitions.push(s_0, V)$
2: while $Partitions \neq \emptyset$ do
3:     $(s, Partition) \leftarrow Partitions.pop()$
4:     if $\|Partition\| > MEMORY\_CAPACITY$ then
5:         MAP$(s, Partition)$
6:     else
7:         NESTED-DFS$(s, Partition)$
8:     FINDPARTITIONS$(Partition)$
9: return No accepting cycle!
```

**Algorithm 5.** MAP$(seed, Partition)$

```
1: $Open.push(\langle seed, \perp \rangle)$
2: $Closed \leftarrow \{seed\}$
3: $Closed.setMap(seed, \perp)$
4: while OPENISNOTEMPTY() do
5:     $(s, propagate) \leftarrow Open.pop()$
6:     for all $t \in$ GETSUCCESSORS$(s)$ do
7:         STOREORCOMBINE$(t, propagate)$
```

**Algorithm 6.** STOREORCOMBINE$(s, map)$

```
1: if $s \in Candidates$ then
2:     $map' \leftarrow Candidates.getMap(s)$
3:     $Candidates.setMap(s,$ MAX$(map, map'))$
4: else
5:     if MEMORYISFULL() then
6:         MERGE()
7:     $Candidates \leftarrow Candidates \cup \{s\}$
8:     $Candidates.setMap(s, map)$
```

In procedure STOREORCOMBINE, we ensure that the currently highest known accepting predecessor for vertex $s$ is stored: if some accepting predecessor for $s$ has been encountered already, we compare it to *map* and store the higher one. Otherwise, we store $s$ and *map*, possibly first making enough memory available through the MERGE operation.

The MERGE procedure joins data stored in internal memory with an external repository. To identify which vertices are new or having their maximal accepting predecessor updated, the procedure traverses all vertices stored on disk and checks whether they are present in the candidate set. For each vertex found in internal memory, MERGE also compares whether the newly found accepting predecessor is higher than the one stored on disk.

Finally, new vertices and vertices with updated accepting predecessor are appended to the open set. New vertices are added to the *Closed* repository, too.

**Algorithm 7.** MERGE

```
1: $Candidates \leftarrow Candidates \cap Partition$ {Intersection is made trivially by a single traversal
   across $Partition$}
2: $Updated = \emptyset$
3: for all $s \in Closed$ do
4:     if $s \in Candidates$ then
5:         UPDATEMAP($s$)
6: for all $s \in Candidates$ do
7:     $Open.push(\langle s, Candidates.getMap(s)\rangle)$
8: $New \leftarrow Candidates \setminus Updated$
9: $Closed \leftarrow Closed \cup New$
10: $Candidates \leftarrow \emptyset$
```

Procedure UPDATEMAP is called from MERGE to compare accepting predecessors of a given vertex $s$, this time taking *all* known information into account. We compare the (so far) highest accepting predecessor for $s$ stored with the candidate set (in memory), the closed set (on disk), and $s$ itself if it is an accepting vertex. Out of those, the maximal vertex (w.r.1t. $\prec$) is stored as new accepting predecessor for $s$.

We discard $s$ from memory if its accepting predecessor stored with the candidate set is not higher than the one stored with the closed set, as it does not yield any useful information.

After the loop between lines 3–5 in MERGE finishes, the candidate set contains only new vertices and vertices whose accepting predecessor in memory has been greater than the one stored on disk. In addition, we return the set of vertices whose accepting predecessor has been changed.

**Algorithm 8.** UPDATEMAP($s$)

```
1: $map \leftarrow Candidates.getMap(s)$
2: $map' \leftarrow Closed.getMap(s)$
3: if ISACCEPTING($s$) then
4:     $map' \leftarrow \text{MAX}(map', s)$
5: $Closed.setMap(s, \text{MAX}(map, map'))$
6: $Candidates.setMap(s, \text{MAX}(map, map'))$
7: if $map \succ map'$ then
8:     if $s = map$ then
9:         exit Accepting cycle found!
10:    $Updated \leftarrow Updated \cup \{s\}$
11: else
12:    $Candidates \leftarrow Candidates \setminus \{s\}$
13: return $Updated$
```

Procedure FINDPARTITIONS is called from DETECTACCEPTINGCYCLE to identify new sub-partitions in a given partition. Therefore, the procedure sorts vertices in the partition by their $map_G$ values. After that, it only traverses the sorted list of vertices sequentially (loop 4–13) to find the beginning and end of partitions, and also to find the maximal accepting predecessor in the given partition, which is from this moment on regarded to be non-accepting (see line 3 of MAP). Note that the condition on line 6 leaves out a partition with $map_G$ value set to $\perp$, since it does not contain any accepting vertex and thus cannot contain an accepting cycle.

**Algorithm 9.** FINDPARTITIONS(*Partition*)

```
1: $Partition.sortByMap()$
2: $newPartition \leftarrow \emptyset$
3: $lastMap \leftarrow \perp$ {$\perp$ is the lowest possible value }
4: for all $(s, map, order) \in Partition$ do
5:     if $lastMap \neq map$ then
6:         if $lastMap \neq \perp$ then
7:             $Partitions.push(seed, newPartition)$
8:         $newPartition \leftarrow \emptyset$
9:     else
10:        $newPartition \leftarrow newPartition \cup \{s\}$
11:    if $map = order$ then
12:        $seed \leftarrow s$
13:    $lastMap \leftarrow map$
14: $Partitions.push(seed, newPartition)$ {Adding last partition}
```

MAP is a revisiting resistant algorithm because it simply traverses the state space and updates $map_G$ values, which are computed as maximum of the values propagated to it. The order and repetition of vertices does not matter, as the maximum stays the same. Henceforward we will refer to the revisiting resistant version of the I/O efficient MAP algorithm as MAP-rr.

Changes in the algorithm are analogous to the modification of the reachability algorithm presented in Sec. 3, but we have to take care of accepting vertices in a special way: between lines 11 and 12 of function UPDATEMAP we put another line:

$$\textbf{if } map' = \text{ORDERNUMBER}(s) \textbf{ then } Updated \leftarrow Updated \cup \{s\}$$

## 5 Complexity Analysis and Comparison

A widely accepted model for the complexity analysis of I/O algorithms is the model of Aggarwal and Vitter [1], in which the complexity of an I/O algorithm is measured solely in terms of the numbers of external I/O operations. This is motivated by the fact that a single I/O operation is approximately six orders of magnitude slower than a computation step performed in main memory [18]. Therefore, an algorithm that does not perform the optimal amount of work but has lower I/O complexity may be faster in practice, when compared to an algorithm that performs the optimal amount of work, but has higher I/O complexity. The complexity of an I/O algorithm in the model of Aggarwal and Vitter is further parametrized by $M$, $B$, and $D$, where $M$ denotes the number of items that fits into the internal memory, $B$ denotes the number of items that can be transferred in a single I/O operation, and $D$ denotes the number of blocks that can

**Table 1.** I/O complexity of algorithms for both modes of candidate set storage. Parameter $p_{max}$ denotes the longest path in the graph going through trivial strongly connected components (without self-loops), $l_{SCC}$ denotes the length of the longest path in the SCC graph, $h_{BFS}$ denotes the height of its BFS tree, and $d$ denotes the diameter of the graph.

| Algorithm | Worst-case I/O Complexity |
|---|---|
| **Candidate set in main memory** | |
| EJ' | $O((l+\|F\|\cdot\|E\|/M)\cdot scan(\|F\|\cdot\|V\|))$ |
| OWCTY | $O(l_{SCC}\cdot(h_{BFS}+\|p_{max}\|+\|E\|/M)\cdot scan(\|V\|))$ |
| MAP | $O(\|F\|\cdot((d+\|E\|/M+\|F\|)\cdot scan(\|V\|)+sort(\|V\|)))$ |
| **Candidate set in external memory** | |
| EJ | $O(l\cdot scan(\|F\|\cdot\|V\|)+sort(\|F\|\cdot\|E\|))$ |
| OWCTY' | $O(l_{SCC}\cdot((h_{BFS}+\|p_{max}\|)\cdot scan(\|V\|)+sort(\|E\|)))$ |
| MAP' | $O(\|F\|\cdot((d+\|F\|)\cdot scan(\|V\|)+sort(\|F\|\cdot\|E\|)))$ |

be transferred in parallel, i.e., the number of independent parallel disks available. The abbreviations $sort(n)$ and $scan(n)$ stand for $\Theta(N/(DB)\log_{M/B}(N/B))$ and $\Theta(N/(DB))$, respectively. In this section we give the I/O complexity of our algorithm and compare it with the complexity of the algorithm by Edelkamp and Jabbar [3].

**Theorem 1.** *The I/O complexity of algorithm* DETECTACCEPTINGCYCLE *is*

$$O(|F|\cdot((d+|E|/M+|F|)\cdot scan(|V|)+sort(|V|)))$$

*where d is the diameter of a given graph.*

*Proof.* Since each partition is identified by its maximal accepting vertex, at most $|F|$ partitions can be found during traversal. Thus, lines 2–8 in DETECTACCEPTINGCYCLE are repeated at most $|F|$ times, and consequently, procedures MAP and FINDPARTITIONS are called at most $|F|$ times as well. Each call of FINDPARTITIONS costs at most $O(scan(|V|)+sort(|V|))$, because of the dominating sort operation on line 1 and the linear scan in loop 4–13. The I/O complexity of MERGE is $O(scan(|V|))$, because it is dominated by the scan operation across the closed set (loop 3–5) and writing of new and updated vertices to the open set.

There are two main sources of I/O operations in procedure MAP: merge operations and open set manipulation. MERGE is indirectly called on lines 4 and 7. It is called whenever the memory becomes full (at most $|E|/M$ times) or the open set becomes empty (at most $d$ times). Reading of *Open* on line 5 costs at most $O(scan(|F|\cdot|V|)) = O(|F|scan(|V|))$, because each vertex can appear in the open set as many times as its associated accepting predecessor changes. □

For the purpose of comparison, we denote our new algorithm as MAP, the algorithm proposed in [4] as OWCTY and the algorithm of Edelkamp and Jabbar [3] as EJ. MAP and OWCTY store the candidate set internally, while EJ stores it externally by default. In the case the candidate set is sorted externally, it is possible to perform the merge operation on a BFS level independently of the size of the main memory. This approach

**Table 2.** Partitions after the first iteration of MAP algorithm. Maximums are taken over partitions with some accepting vertex in them.

| Experiment | Graph Size | Number of Partitions | Max. Partition Size | | Vertices with $map_G = \perp$ | |
|---|---|---|---|---|---|---|
| Lamport(5),P4 | 74,413,141 | 838,452 | 454,073 | $< 1\%$ | 38,717,846 | 52% |
| MCS(5),P4 | 119,663,657 | 3,373,145 | 108,092 | $< 1\%$ | 60,556,519 | 51% |
| Peterson(5),P4 | 284,942,015 | 11,451 | 12,029,114 | 4% | 142,471,098 | 50% |
| Phils(16,1),P3 | 61,230,206 | 336,339 | 129,023 | $< 1\%$ | 43,046,721 | 70% |
| Rether(16,8,4),P2 | 31,087,573 | 33,353 | 5 | $< 1\%$ | 30,920,813 | 99% |
| Szymanski(5),P4 | 419,183,762 | 20,064 | 131,441,308 | 31% | 209,596,444 | 50% |

is suitable for those cases where memory is small, or the graph is orders of magnitude larger. A disadvantage of the approach is the need to sort during each merge operation. Furthermore, it cannot be combined with heuristics, such as Bloom filters and a lossy hash table [12]. Fortunately, all three algorithms are modular enough to be able to work in both modes. Tab. 1 shows the I/O complexities for all three algorithms in both variants.

It can be seen that the upper bound for the complexity of MAP is worse than the one of EJ and OWCTY (in both modes of the candidate set). Nevertheless, we claim that the complexity is reasonable in most cases, for a number of reasons. First, the algorithm usually performs at most two iterations of the loop between lines 2–8 in procedure DETECTACCEPTINGCYCLE: if an accepting cycle was not found during the first iteration, the state space is partitioned into many partitions (as shown in Tab. 2). Furthermore, if the state space was partitioned evenly, then 1000 partitions would be enough to divide a 1 Terabyte state space into blocks sufficiently small to fit into very modestly sized internal memory, by today's standards. Even if some partitions would not fit into main memory yet, another partitioning round usually decreases the maximal partition size enough such that all remaining partitions fit into internal memory. Therefore, it is reasonable to expect that the algorithm becomes fully internal after a very small number of iterations.

Second, $d$ is commonly not proportional to the size of the state space and is usually not much higher than $h_{BFS}$.

Third, the upper bound $|F| \cdot |V|$ on the number of vertices revisited due to updates of a $map_G$ value is quite coarse. We have measured the amount of $map_G$ updates for the MAP algorithm with a reverse-BFS ordering of vertices. We found that *map* updates take commonly not more than 20% of the graph exploration (see Tab. 4).

Taking this into account, the complexity of MAP could be very close to

$$O((h_{BFS} + |E|/M) \cdot scan(|V|) + sort(|V|))$$

in most practical cases. We note that this equals the complexity of I/O efficient reachability plus sorting the set of vertices. Our measurements confirm this claim, as shown in Tab. 4.

**Table 3.** Comparison of revisiting techniques and simple I/O efficient reachability

| Experiment | Normal (hours) | Revisiting Resistant (hours) | |
|---|---|---|---|
| Lamport(5),P4 | 02:51:09 | 01:19:32 | 46% |
| MCS(5),P4 | 03:56:26 | 02:41:45 | 68% |
| Peterson(5),P4 | 19:38:32 | 09:02:37 | 46% |
| Phils(16,1),P3 | 02:09:45 | 01:41:24 | 77% |
| Rether(16,8,4),P2 | 13:54:29 | 00:29:19 | 3% |
| Szymanski(5),P4 | 51:20:32 | 17:54:14 | 34% |
| On average | 100% | | 46% |

**Table 4.** Run times of reachability and MAP algorithm

| Model | Reachability (hours) | MAP (hours) | |
|---|---|---|---|
| Lamport | 2:51:09 | 3:12:09 | 112% |
| MCS | 3:56:26 | 4:28:06 | 113% |
| Phils | 2:09:45 | 2:29:26 | 115% |

## 6 Experiments

In order to obtain experimental evidence about the behavior of our algorithm in practice, we implemented an I/O efficient reachability procedure and three I/O efficient LTL model checking algorithms.

All algorithms have been implemented on top of the DiVinE library [19], providing the state space generator, and the STXXL library [20], providing the needed I/O primitives. Experiments were run on 2 GHz Intel Xeon PC, the main memory was limited to 2 GB, the disk space to 60 GB and wall clock time limit was set to 120 hours. Algorithm MAP-rr is a variant of MAP exploiting its revisiting resistance. Algorithm EJ was implemented as a procedure that performs the graph transformation as suggested in [3] and then employs I/O efficient breadth-first search to check for a counter example. Note, that our implementation of [3] does not include the $A^*$ heuristics and hence can be less efficient when searching for an existing counter example. The procedure is referred to as *Liveness as Safety with BFS* (LaS-BFS) [21].

First of all, we have measured the impact of revisiting resistance on procedure Reachability. We have obtained results that demonstrate significant speed-up, as shown in Tab. 3. We have also measured run times and memory consumption of LaS-BFS, OWCTY, MAP and MAP-rr. The experimental results are listed in Tab. 5. We note that just before the unsuccessful termination of LaS-BFS due to exhausting the disk space, the BFS level size still tended to grow. This suggests that the computation would last substantially longer if sufficient disk space would have been available. For the same input graphs, algorithms OWCTY, MAP and MAP-rr manage to perform the verification using a few Gigabytes of disk space only. All the models and their LTL properties are taken from the BEEM project [22].

Measurements on models with valid properties demonstrate that MAP is able to successfully prove their correctness, while LaS-BFS fails. Additionally, MAP's

**Table 5.** Run times in `hh:mm:ss` format and memory consumption on a single workstation. "OOS" means "out of space".

| | LaS-BFS | | OWCTY | | MAP | | MAP-rr | |
|---|---|---|---|---|---|---|---|---|
| Experiment | Time | Disk | Time | Disk | Time | Disk | Time | Disk |
| **Valid Properties** | | | | | | | | |
| Lamport(5),P4 | (OOS) | | 02:37:17 | 5.5 GB | 03:16:36 | 5.7 GB | 02:37:56 | 8.5 GB |
| MCS(5),P4 | (OOS) | | 03:27:05 | 9.8 GB | 04:59:17 | 10 GB | 04:13:21 | 11 GB |
| Peterson(5),P4 | (OOS) | | 18:20:03 | 26 GB | 25:09:35 | 26 GB | 15:24:29 | 27 GB |
| Phils(16,1),P3 | (OOS) | | 01:49:41 | 6.2 GB | 02:31:33 | 7.8 GB | 02:19:20 | 8.1 GB |
| Rether(16,8,4),P2 | 53:06:44 | 12 GB | 07:22:05 | 3.2 GB | 12:31:18 | 6.3 GB | 00:39:07 | 6.3 GB |
| Szymanski(5),P4 | (OOS) | | 45:52:25 | 38 GB | 59:35:25 | 38 GB | 29:09:12 | 39 GB |
| **Invalid Properties** | | | | | | | | |
| Anderson(5),P2 | 00:00:17 | 50 MB | 07:14:23 | 3.3 GB | 00:00:07 | 2 MB | 00:00:01 | 4 MB |
| Bakery(5,5),P3 | 00:25:59 | 5.4 GB | 68:23:34 | 38 GB | 00:00:09 | 16 MB | 00:00:23 | 54 MB |
| Szymanski(4),P2 | 00:00:50 | 203 MB | 00:20:07 | 253 MB | 00:00:04 | 2 MB | 00:00:02 | 4 MB |
| Elevator2(7),P5 | 00:01:02 | 130 MB | 00:00:25 | 6 MB | 00:00:05 | 2 MB | 00:00:01 | 3 MB |

performance does not differ much from the performance of OWCTY. Moreover, with the use of revisiting resistant techniques, MAP-rr is able to outperform OWCTY in many cases. We observe that specifically in cases with high $h_{BFS}$—e. g., Rether(16,8,4),P2—time savings are substantial.

A notable weakness of OWCTY is its slowness on models with invalid properties. It does not work on-the-fly, and is consequently outperformed by LaS-BFS in the aforementioned class of inputs. Algorithms MAP and MAP-rr do not share OWCTY's drawbacks, and in fact they outperform both, OWCTY and LaS-BFS on those inputs. This can be attributed to their on-the-fly nature: On all our inputs, a counter example, if existing, is found during the first iteration.

## 7 Conclusions

We described *revisiting resistance*, a distinct property of graph algorithms, and showed how it can be of practical use to the I/O efficient approach of processing very large graphs. In particular, we described how a simple I/O efficient reachability procedure with delayed duplicate detection can be extended to exploit its revisiting resistance and showed that the extension is valuable in practice. Furthermore, we analyzed existing I/O efficient algorithms for LTL model checking and showed that the OWCTY algorithm is not revisiting resistant. We introduced a new I/O efficient revisiting resistant algorithm for LTL model checking that employs the *Maximal Accepting Predecessor* function to detect accepting cycles. We analyzed the I/O complexity of the new algorithm, and showed that due to the revisiting resistance, the algorithm exhibits competitive runtimes for verification of valid LTL properties while preserving its on-the-fly nature. According to our experimental results, the algorithm outperforms other I/O efficient algorithms on invalid LTL properties even if it is being slowed down with the vertex revisiting.

## References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proc. of LICS 1986, pp. 332–344. Computer Society Press (1986)
3. Edelkamp, S., Jabbar, S.: Large-Scale Directed Model Checking LTL. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 1–18. Springer, Heidelberg (2006)
4. Barnat, J., Brim, L., Šimeček, P.: I/O Efficient Accepting Cycle Detection. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 281–293. Springer, Heidelberg (2007)
5. Korf, R.E., Schultze, P.: Large-Scale Parallel Breadth-First Search. In: Proc. of AAAI, pp. 1380–1385. AAAI Press / The MIT Press (2005)
6. Korf, R.E.: Best-First Frontier Search with Delayed Duplicate Detection. In: Proc. of AAAI, pp. 650–657 (2004)
7. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
8. Munagala, K., Ranade, A.: I/O-complexity of graph algorithms. In: Proc. of SODA, Society for Industrial and Applied Mathematics, pp. 687–694 (1999)
9. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
10. Korf, R.E.: Best-First Frontier Search with Delayed Duplicate Detection. In: Proc. of AAAI, pp. 650–657. AAAI Press / The MIT Press (2004)
11. Zhou, R., Hansen, E.A.: Structured Duplicate Detection in External-Memory Graph Search. In: Proc. of AAAI, pp. 683–689 (2004)
12. Hammer, M., Weber, M.: To Store or Not To Store. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 51–66. Springer, Heidelberg (2007)
13. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
14. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)
15. Ravi, K., Bloem, R., Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 143–160. Springer, Heidelberg (2000)
16. Brim, L., Černá, I., Moravec, P., Šimša, J.: How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In: Proc. of PDMC (2005)
17. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: The SPIN Verification System, American Mathematical Society, pp. 23–32 (1996)
18. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. 33(2), 209–271 (2001)
19. Barnat, J., Brim, L., Černá, I., Šimeček, P.: DiVinE – The Distributed Verification Environment. In: Proc. of PDMC, pp. 89–94 (2005)
20. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL Data Sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)
21. Schuppan, V., Biere, A.: Efficient Reduction of Finite State Model Checking to Reachability Analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2–3), 185–204 (2004)
22. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)

2009 IEEE/ACM International Conference on Automated Software Engineering

# Cluster-Based I/O-Efficient LTL Model Checking

Jiří Barnat, Luboš Brim, Pavel Šimeček
*Faculty of Informatics, Masaryk University*
*Brno, Czech Republic*
*Email: {brim,barnat,xsimece1}@fi.muni.cz*

***Abstract*—I/O-efficient algorithms take the advantage of large capacities of external memories to verify huge state spaces even on a single machine with low-capacity RAM. On the other hand, parallel algorithms are used to accelerate the computation and their usage may significantly increase the amount of available RAM memory if clusters of computers are involved. Since both the large amount of memory and high speed computation are desired in verification of large-scale industrial systems, extending I/O-efficient model checking to work over a network of computers can bring substantial benefits. In this paper we propose an explicit state cluster-based I/O efficient LTL model checking algorithm that is capable to verify systems with approximately $10^{10}$ states within hours.**

***Keywords*-I/O efficient algorithms; LTL; model checking; parallel algorithms**

## I. Introduction

An importance of automated formal verification grows as modern hardware and software systems are becoming more complex and demands on their reliability increases. Several kinds of verification techniques, model checking in particular, have already been successfully employed to help to handle this difficult task. However, techniques for automated formal verification are computationally demanding and memory-intensive in general and their applicability to extremely large and complex systems routinely seen in practice these days is limited. To efficiently handle large industrial systems scalable methods with moderate run-time and memory requirements are required.

As a matter of fact, verification and analysis methods that focus on efficient employment of increased amount of available computational power are subject of extensive research. These are, for example, techniques to fight memory limits with efficient utilization of external I/O devices [12], [7], [6], [3], techniques that introduce cluster-based algorithms to employ aggregate power of network-interconnected computers [1], [4], or techniques to speed-up the verification on multi-core processors or graphics processing units.

In this paper we focus on explicit state LTL model checking. Recent achievements in parallel LTL model checking have allowed to verify quite large systems within times that are comparable to sequential verification of much smaller systems. On the other hand, I/O efficient approach to LTL model checking is ready to handle even larger state spaces. However, the time needed to complete the verification is typically hours. Speeding up I/O efficient LTL model checking by using parallel architectures is therefore a natural way, how to make I/O efficient approach more acceptable from the practical point of view. The research on *parallel I/O efficient* verification started just recently. In [10], the authors designed an algorithm for parallel breadth-first search on general graphs stored explicitly on a disk. However, in model checking, duplicate detection is performed differently due to implicit definition of the graph. Nevertheless, the ideas behind parallelization have been successfully used in [9], where the authors present an algorithm for a parallel I/O-efficient implementation of the search over a system state space represented as *undirected* graph. Experiments provided in [9] are on relatively small state spaces and also the implementation has poor performance considering the size of state spaces. Another approach is [8]. The main advantages gained from streaming the state space is easy load balancing and good scalability achieved by better memory locality. On the other hand, the approach suffers from potential existence of thin parts (substantially slower computers) and the fact that in the end all nodes store the entire state space.

We propose a new approach for explicit state LTL model checking that combines I/O efficient techniques with parallel distributed-memory paradigm (like clusters of workstations). Compared to existing algorithms, our algorithm is able to work on *directed* graphs representing state spaces and uses a classical hash-based partitioning mechanism (the owner of each system state is determined using a hash function).

## II. Reachability Analysis

Reachability analysis searches for system states other than allowed ones in order to disprove system correctness. From the algorithmic point of view, the reachability analysis is based on a simple graph traversal procedure, such as breadth-first search. Since a graph representing a state space is given implicitly – by an initial state and a successor function – the algorithm stores all visited states in so called *Closed* set. Another data structure used in graph traversal is the *Open* set – the set of visited, but not yet traversed system states. The algorithm repeatedly removes states from the *Open* set and traverses them, i.e. let them evolve into possible succeeding states. When a state is generated, it is checked against visited states whether it is new or not. If it has been visited before, i.e. it is found either in *Open* or *Closed* set, it is discarded

**Algorithm 1** *ReachabilityAnalysis*

```
1: $Candidates \leftarrow \emptyset$, $Closed \leftarrow \langle s \rangle$
2: $Open \leftarrow \langle s \rangle$, $Open' \leftarrow \langle \rangle$
3: $s \leftarrow GetInitialVertex()$
4: while $Open \neq \langle \rangle$ do
5:     $s \leftarrow Open.pop()$
6:     for all $t \in GetSuccessors(s)$ do
7:         $owner \leftarrow Hash(t) \mod CPU\_COUNT$
8:         if $owner = NETWORK\_ID$ then
9:             if $t \notin Candidates$ then
10:                $Candidates \leftarrow Candidates \cup \{t\}$
11:                if $CandidatesTooLarge()$ then
12:                    $Merge()$
13:        else $sendState(t, owner)$
14:    if $Open = \langle \rangle$ then
15:        $Merge()$
16:        $Open \leftarrow Open'$
17:        $Open' \leftarrow \langle \rangle$
```

**Algorithm 2** *Merge*

```
1: $Synchronize()$
2: for all $s \in Closed$ do
3:     if $s \in Candidates$ then
4:         $Candidates \leftarrow Candidates \setminus \{s\}$
5: for all $s \in Candidates$ do
6:     $Open'.push(s)$
7:     $Closed \leftarrow Closed \cup \{s\}$
8: $Candidates \leftarrow \emptyset$
9: $Synchronize()$
```

immediately. New states are inserted into the *Open* set. The traversal algorithm is initiated by inserting initial states into the *Open* set, and terminates as soon as there are no states in the *Open* set waiting for traversal.

A simplified pseudo-code of the algorithm for parallel reachability analysis with external memory is given in Algorithm 1. It utilizes the fact that the correctness of reachability analysis is not influenced by the order in which states are visited, hence the states may be processed in parallel. Moreover, checking whether a given state has already been visited may be postponed and performed in a big batch. *Candidates* is the set of generated states whose check against the *Closed* set is postponed until the next MERGE operation happens. The work-flow now is as follows. A visited, but not yet traversed state is extracted form the *Open* set and the succeeding states are generated using GETSUCCESSORS function. When a system state is generated, it is first checked for the ownership. If the state is owned by other computation node than the local one, i.e. the node that generated the state, the state is sent over the network to its owner using SENDSTATE function. Otherwise, the state is checked against the set of *Candidates*, and inserted into this set if not yet there. Configurations that are received from other computation nodes are processed as if they were generated locally, i.e., they are inserted into the *Candidates* set upon their reception. Once the *Candidates* set gets too large to fit into the internal memory, the MERGE operation is performed. Note that in our approach, the *Candidates* set is stored internally [1], [3]. However, there is an alternative approach to the delayed duplicate detection where also the *Candidates* set is stored externally [9].

The MERGE operation should be also performed each time the *Open* set becomes empty (line 15 of Algorithm 1) to identify new states in *Candidates* and copy them to the *Open* set. Since MERGE operations are quite expensive when the *Closed* set becomes large, we also apply a heuristic called *merge omission*, which estimates, whether it would be faster to perform MERGE or continue a traversal without check for duplicates – in such a case all states from *Candidates* are simply moved to the *Open* set, including duplicates. The heuristic is for simplicity reasons not involved in the pseudo-code and details can be found in [3]. Within the MERGE operation, whose pseudo-code is given in Algorithm 2, all the states found in the *Closed* set are removed from *Candidates* set at first. Note that this operation requires traversing the *Closed* set stored in the external memory.

## III. MODEL CHECKING LTL PROPERTIES

Reachability analysis covers only very limited class of system properties. For more sophisticated analysis, properties have to be written down in a richer logical framework, such as Linear Temporal Logic (LTL).

We have built our parallel I/O efficient approach upon an existing parallel LTL model checking algorithm called OWCTY (One Way Catch Them Young) [5]. The reason was that the algorithm that in our earlier work we have proposed and I/O efficient algorithm that builds on the same principle [2]. The main idea behind the OWCTY algorithm stems from the fact that a directed graph can be topologically sorted if and only if it is acyclic. The core of the cycle detection algorithm is thus an application of the standard linear topological sort algorithm to the input graph. Failure in topologically sorting the graph means that the graph contains a cycle. Accepting cycles are detected with multiple iterations, each topologically sorting a subset of vertices. Every iteration consists of reachability and elimination procedures. The reachability procedure removes vertices unreachable from accepting vertices (as these cannot belong to an accepting cycle) and computes indegrees of all remaining vertices. The succeeding elimination procedure recursively removes vertices whose predecessor count drops to zero (following the topological sort procedure). An accepting cycle is detected if there is a set of states that cannot be further reduced by more iterations. If all vertices are successively removed, no accepting cycle is present in the graph.

The I/O efficient implementation consists of I/O efficient parallel implementations of both reachability and elimination procedures. The reachability procedure is very similar to the one described in Section II, while the elimination procedure can be obtained from the reachability algorithm by enqueuing only states with zero indegrees. Both procedures are restricted only to vertices that have not been removed yet. Merge omission heuristic is not allowed because of the need to record indegrees of vertices precisely.

## IV. Experimental Results

To demonstrate that our approach results in significant improvements we have performed a set of experiments that are described in this Section. Our experimental setting includes different workstation configurations:

| |
|---|
| *Configuration 1* (8 machines)<br>Intel Pentium 4 2.80 GHz (1 core), 2 GB RAM, 3× HDD Seagate Barracuda 7200.10 160 GB, ext3 fs |
| *Configuration 2* (18 machines)<br>2×Intel Xeon 4 2 GHz (2 cores), 16 GB RAM, HDD Fujitsu MAY2073RC 73 GB, ext3 fs |
| *Configuration 3* (1 machine)<br>AMD Phenom II 3 GHz (4 cores), 8 GB RAM, 4× HDD Seagate Barracuda 7200.10 160 GB, ext3 fs |

For this experimental study we use models from the BEEM database [11] – Table I. The size of *Candidates* set is limited to 1 GB per process in order to demonstrate usability even for systems with a small amount of RAM.

First, we show that aggregate external memory of workstations in a small cluster may render a meaningful computing platform with hundreds or thousands of Gigabytes of memory. Table II shows run times of state space generation algorithm on 12, 15 and 18 machines in config. 2. It is apparent that the algorithm benefits from parallel processing and on 18 processors it runs significantly faster than on smaller number of workstations. On 18 workstations it is possible to enumerate $2 \cdot 10^{10}$ states (model Peterson(6)) within approximately one day. Using this number of workstations, such a large state space would not fit in the main memory even if each workstation utilized all 16 GB RAM available.

To provide a better idea of algorithm scalability over a lower number of workstations we also provide scalability measurements on smaller number of machines and smaller state spaces – Figure 1. From the achieved results it is apparent that the algorithm scales sub-linearly. The sub-linearity has two main reasons:

- merge omission heuristic (with less memory less states can be revisited and so less merge operation are saved),
- some BFS levels are too large to fit in RAM, which also increases a total number of merge operations.

The fact that the amount of available RAM memory influences a count of I/O operations, blurs how efficient is a workload distribution itself. Therefore, we provide also scalability results for a very artificial scenario where the sum of memory allocated for a candidate set is kept constant and revisiting heuristic is not applied. For simplicity, we provide results for input Elevator(7,5,1) only – Figure 2. We have observed very good scalability with very small overhead, since I/O operations dominate over a communication time.

Another level of parallelization is a usage of multiple disks in each machine. In our experiment, we compare the performance of 8 workstations with 1 disk to 4 workstations with 2 disks. We have chosen models with three smallest state spaces from a test set, because of a smaller number of workstations in config. 2. For every input, Figure 3 shows run times on both platforms. In case of the state space for Lann(5,0,1,0) that was generated on 8 workstations, there were no I/O operations performed at all, since the state

Figure 2. Scalability measurements independent of a RAM size on Elevator(7,5,1)

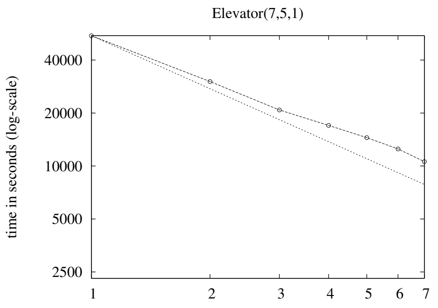


Table I
Models used in experiments

| | # of states | \|state space\| | BFS levels |
|---|---|---|---|
| Peterson(6) | 22,800,638,886 | 728 GB | 121 |
| Cambridge(64,1,1) | 2,243,706,411 | 316 GB | 867 |
| MCS(6) | 9,045,324,641 | 267 GB | 50 |
| Public Subscribe(3,1) | 1,153,014,089 | 52 GB | 166 |
| Elevator(7,5,1) | 550,895,416 | 35 GB | 131 |
| Lann(5,0,1,0) | 160,025,986 | 5 GB | 359 |
| Elevator(7,5,1)-Prop3 | 614,962,062 | 39 GB | 174 |
| Lann(5,0,1,0)-Prop3 | 320,045,746 | 11 GB | 367 |
| MCS(6)-Prop4 | 17,908,781,904 | 550 GB | 52 |

(Lower three models are automata products of systems and Büchi automata representing verified properties)

Table II
Run times in hh:mm format and ratios of workload.

| | Cambridge(64,1,1) | | | MCS(6) | | |
|---|---|---|---|---|---|---|
| Nodes | Time | Disk | # Merge | Time | Disk | # Merge |
| 12 | 22:16 | 81 % | 224 | 14:13 | 77 % | 112 |
| 15 | 16:54 | 78 % | 194 | 10:09 | 72 % | 88 |
| 18 | 12:55 | 70 % | 156 | 07:43 | 69 % | 76 |

| | Peterson(6) | | |
|---|---|---|---|
| Nodes | Time | Disk | # Merge |
| 12 | 60:40 | 90 % | 254 |
| 15 | 39:34 | 84 % | 196 |
| 18 | 27:22 | 80 % | 159 |

Figure 1. Scalability measurements on Elevator(7,5,1), Lann(5,0,1,1,0) and Public Subscribe(3,1)

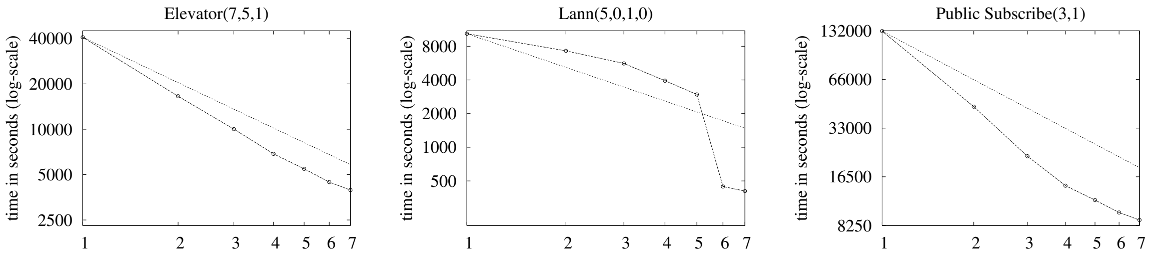
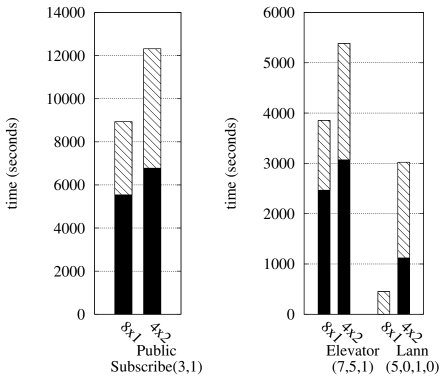


Figure 3. Run times of distributed state space generation on three models in two different settings

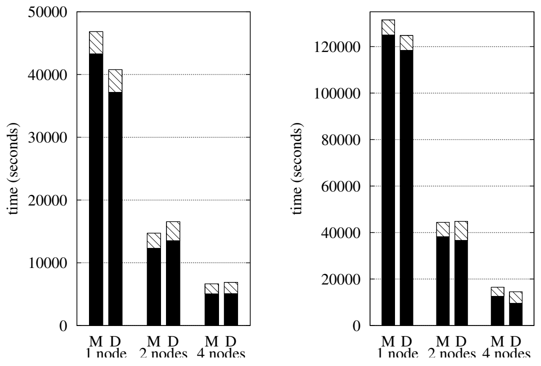


Numbers of merge operations:

| | 8x1 | 4x2 |
|---|---|---|
| Public Subscribe(3,1) | 58 | 69 |
| Elevator(7,5,1) | 35 | 44 |
| Lann(5,0,1,0) | 0 | 74 |

8x1 = 8 workstations with 1 disk, 4x2 = 4 workstations with 2 disks
Solid black boxes stand for a time spent on disk operations.

Figure 4. Distributed I/O-efficient state space generation – comparison of multi-core (M) and distributed (D) computation on 1, 2 and 4 cores and a corresponding number of workstations



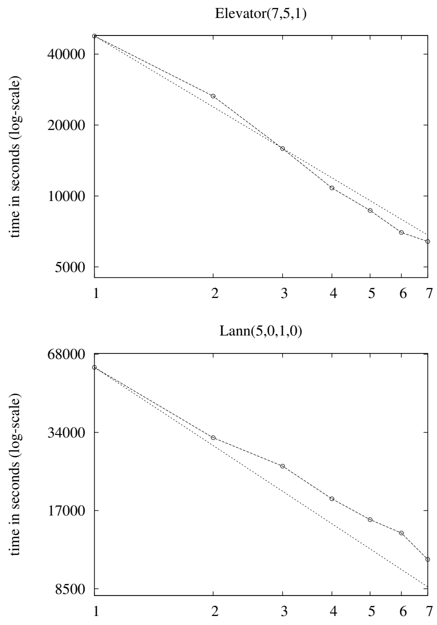
Solid black boxes stand for a time spent on disk operations

space fit in the aggregate internal memory. For the other two models, slightly higher I/O times were achieved. Although the measured overall disk transfer rate on one workstation has been almost double the rate of a single disk, more merge operations were performed on 4 workstations due to the smaller aggregate RAM. Hence, the merge omission heuristic was not applied as often as on 8 workstations. Some additional merge operations in 4-workstation setting have been caused by exhausting the main memory. Moreover, lower number of processors involved in the state space generation lead to almost doubling the time spent on other than disk operations. Nevertheless, the setting with two disks has still significantly better performance/price ratio, since a price of one hard disk is much lower than a price of a single workstation.

Since I/O-efficient model checking mostly brings low-cost solutions to enormously large problems, there is only a little experience with running it on multi-processor systems. Figure 4 demonstrates that our distributed state space generator works well even on a multi-core system without any system-specific modifications. We compared distributed and multi-core run of the same tasks resulting in similar scaling on both architectures. We used different system configuration for multi-core (config. 3) and distributed (config. 2) setting – this is where the difference in speed on 1 node (i.e. without communication) comes from. We have also observed that while there is not much difference in time spent on operations other than I/O between system config. 2 and 3 on a single node, on more nodes multi-core computations profit from faster communication.

To test how LTL model checking works in a distributed I/O efficient setting, we measured run times on two relatively small and one huge input (see Table I for sizes). For the experiments we used 18 workstations config. 2:

Figure 5. Scalability of distributed I/O efficient OWCTY on two inputs



| | Time | *Init-ASet* | *Elim-No-Acc.* | *Elim-No-Pred.* |
|---|---|---|---|---|
| Elevator(7,5,1)-Prop3 | 00:49 | 90 % | 6 % | 4% |
| Public Subscribe(3,1)-Prop3 | 03:59 | 74 % | 22 % | 4% |
| MCS(6)-Prop4 | 59:24 | 64 % | 22 % | 14% |

Percentage numbers express a portion of time spent in a given sub procedure. We deduce that initial state space generation is a major time consumer, due to a low number of elimination steps and cheaper operations over the approximation set substantially reduced with elimination. Successful verification of MCS(6) demonstrates the power of distributed I/O efficient model checking. Nevertheless, there is still space for optimization. We hope that together with disk parallelization and storage of the candidate set to disk we could reach fractions of currently measured times.

Since most of verification time is spent in the initial state space generation, it is not very surprising, that OWCTY scales similarly well as reachability analysis – see Figure 5.

## V. Conclusions

This paper presents a novel approach for explicit-state LTL model checking of very large systems. We employed a combination of a distributed-memory approach with I/O efficient usage of external memory. First, we designed parallel state space generator and analyzed its performance in various settings. Then we built LTL model checker upon it. Our algorithm scales well over both a cluster of workstation and a multi-core machine. We are able to generate state spaces and verify systems with more than $10^{10}$ states on a small compute cluster. A unique feature of the algorithm is, that due to merge omission heuristic, it is able to take the advantage of aggregate internal memory in distributed environment and thus obtains sub-linear speed up in some cases.

## Acknowledgment

The work of authors has been supported by Czech Science Foundation grant No. 201/09/P497, grant No. 201/09/1389, and grant No. 102/09/H042. Pavel Šimeček has also been supported by the Academy of Sciences grant No. 1ET408050503.

## References

[1] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[2] J. Barnat, L. Brim, and P. Šimeček. I/O Efficient Accepting Cycle Detection. In *CAV'07*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.

[3] J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In *TACAS'08*, volume 4963 of *LNCS*, pages 48–62. Springer, 2008.

[4] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free $\mu$-Calculus. In *TACAS'01*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

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

[6] S. Edelkamp and S. Jabbar. Large-Scale Directed Model Checking LTL. In *SPIN'06*, volume 3925 of *LNCS*, pages 1–18. Springer, 2006.

[7] M. Hammer and M. Weber. "To Store Or Not To Store" Reloaded: Reclaiming Memory On Demand. In *FMICS'06*, volume 4346 of *LNCS*, pages 51–66. Springer, 2006.

[8] V. Holub and P. Tůma. Streaming state space: A method of distributed model verification. In *TASE'07*, pages 356–368. IEEE Computer Society, 2007.

[9] S. Jabbar and S. Edelkamp. Parallel External Directed Model Checking with Linear I/O. In *VMCAI'06*, volume 3855 of *LNCS*, pages 237–251. Springer, 2006.

[10] R. Korf and P. Schultze. Large-Scale Parallel Breadth-First Search. In *AAAI'05*, pages 1380–1385. AAAI Press/The MIT Press, 2005.

[11] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[12] U. Stern and D. L. Dill. Using Magnetic Disk Instead of Main Memory in the Murphi Verifier. In *CAV'98*, volume 1427 of *LNCS*, pages 172–183. Springer, 1998.

# Local Quantitative LTL Model Checking[⋆]

Jiří Barnat, Luboš Brim, Ivana Černá, Milan Češka, and Jana Tůmová

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,cerna,xceska,xtumova}@fi.muni.cz

**Abstract.** Quantitative analysis of probabilistic systems has been studied mainly from the global model checking point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in local model checking approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. In this paper we present several particular local model checking techniques that if applied to global model checking procedure reduce the runtime needed from days to minutes.

## 1 Introduction

System design techniques employing probability are becoming widely used. They provide designers with reasonably efficient means to break symmetry in the system or to implement randomized algorithms. Probabilistic actions are also used for modeling various nondeterministic aspects such as human unpredictable decisions, occurrence of external stimuli, or simply the presence of hardware errors. As the interest in the probabilistic systems is growing, supported mainly by their potential practical use, there is also increased interest in formal techniques for their analysis and verification, model checking in particular.

There are two different tasks related to model checking over probabilistic systems. Given a formula and probabilistic system, the so called *qualitative* analysis refers to the problem of deciding whether the probabilistic system satisfies the formula with the probability one. On the other hand, the so called *quantitative* model checking refers to the problem of deciding the maximal and minimal probability the given formula is satisfied for the probabilistic system. For model checking linear time properties, the qualitative problem can be solved similarly to the nondeterministic case, i.e. using automata-based approach. The problem reduces to the problem of the detection of an *Accepting End Component* (AEC) in the graph of the underlying product of the probabilistic system and the $\omega$-regular automaton expressing the (negation of) the verified property [22,11].

[⋆] This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

For the quantitative case the model checking procedure is a little bit more complex [3,12]. Similarly to the qualitative case, the probabilistic system is multiplied with the semi-deterministic $\omega$-regular property automaton and all the AECs are identified in its underlying graph. After that, the graph is transformed into a linear programming problem (set of inequalities over states of the probabilistic system and an objective function to be maximized). Every variable in the linear programming instance corresponds to a state in the system in the sense that the value computed for the variable is exactly the maximal probability of satisfaction of the examined property, if the property is evaluated from the particular state. States in an AEC satisfy the examined property with the probability one.

Both qualitative and quantitative analysis of probabilistic systems has been studied mainly from the *global model checking* point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in *local model checking* approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. It is a well-known fact that from practical point of view, the system designers are often interested in the probability of satisfaction of the property for some particular states only (initial state most typically). This is not taken into account in the general global model checking scheme as suggested in [3,12].

There are several software tools performing qualitative and/or quantitative probabilistic model checking. Probably the most established probabilistic model checker is the *symbolic* model checker PRISM [16]. It provides support for automated analysis of a wide range of quantitative properties for three types of probabilistic models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes (MDPs). The property specification language of PRISM incorporates the temporal logics PCTL [15] and CSL [1] as well as extensions for quantitative specifications and costs/rewards. As for *enumerative* approach to model checking, the model checker to be mentioned is LiQuor [10]. LiQuor is capable of verifying probabilistic systems modeled as ProbMeLa programs. ProbMeLa is a probabilistic guarded command language with an operational semantics based on finite MDPs. LiQuor allows qualitative and/or quantitative analysis for $\omega$-regular linear time properties. The tool follows the standard automata-based model checking approach and involves partial order reduction technique for MDPs [4] to fight the state explosion problem. Recently, a parallel enumerative probabilistic model checker – ProbDiVinE, has been released [5]. Likewise LiQuor, ProbDiVinE provides means for verification of quantitative and qualitative linear time properties of MDPs. The unique feature of ProbDiVinE is its capability of employing combined power of multiple CPU cores available on latest hardware systems to solve large verification problems. All the techniques presented in this paper have been implemented and experimentally evaluated using the ProbDiVinE model checker. Yet another tool for verification of MDPs is the model-checker RAPTURE [8]. It employs an automatic abstraction refinement and essential state reduction techniques to fight the state explosion problem [8].

As the main contribution of this paper we introduce several techniques that allow to improve the general *quantitative* verification procedure using the locality of the model checking goal. These local model checking techniques can be applied to the global model checking procedure resulting in a significant speed-up as indicated by our experimental evaluation. In addition, the locality of the techniques supports their natural integration into a parallel tool, giving thus further advantages in terms of speed and scalability.

The rest of the paper is organized as follows. Section 2 states the necessary definitions and recalls the general scheme of quantitative model checking procedure. Section 3 introduces our new techniques to improve the general verification scheme, Section 4 reports on experimental evaluation of these techniques, and Section 5 concludes the paper.

## 2 Preliminaries

In this subsection, we briefly review fundamentals of LTL model checking over finite state probabilistic systems and fix some notation.

### 2.1 Probabilistic Model Checking

*Markov decision processes* (MDPs) are used as the standard modeling formalism for asynchronous probabilistic systems, supporting both nondeterminism and probability. A Markov decision process [13,21,22], is a tuple $M = (S, Act, P, init, AP, L)$, where $S$ is a finite set of states, $Act$ is a finite set of actions, $P : (S \times Act \times S) \to [0, 1]$ is a probability matrix, $init \in S$ is the initial state, $AP$ is a finite set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. $Act(s)$ denotes the set of actions that are enabled in the state $s$, i.e. the set of actions $\alpha \in Act$ such that $P(s, \alpha, t) > 0$ for some state $t \in S$. For any state $s \in S$, we require that $Act(s) \neq \emptyset$ and $\forall \alpha \in Act(s). \sum_{s' \in S} P(s, \alpha, s') = 1$.

An infinite run of an MDP is a sequence $\tau = s_0, \alpha_1, s_1, \alpha_2, \ldots \in (S \times Act)^\omega$ such that $\alpha_i \in Act(s_{i-1})$. A trajectory of $\tau$ is the word $L(s_0), L(s_1), L(s_2), \ldots$ over the alphabet $2^{AP}$ obtained by the projection of $\tau$ to the state labels.

The intuitive operational semantics of an MDP is as follows. If $s$ is the current state then an action $\alpha \in Act(s)$ is chosen nondeterministically and is executed leading to a state $t$ with probability $P(s, \alpha, t)$. We refer to $t$ as an *$\alpha$-successor* of $s$ if $P(s, \alpha, t) > 0$. State $s$ is called *deterministic* if exactly one action is enabled in $s$. If all states of an MDP are deterministic, the MDP is called *Markov chain.* To resolve the nondeterminism of an MDP a *scheduler* function is used. We consider deterministic history dependent schedulers which are given by a function $D$ assigning an action $D(\sigma) \in Act(s_n)$ to every finite run $\sigma = s_0, \alpha_1, \ldots, \alpha_n, s_n$. Given a scheduler $D$, the behavior of $M$ under $D$ can be formalized as a Markov chain.

Let $M$ be a Markov Chain, $s \in S$ be a state of $M$, and $X$ be a set of runs of $M$ originating at $s$. We define *the probability of the set* $X$ as a measure of the set $X$ in the set of all runs of $M$ originating at $s$. A set $X$ of runs of a Markov Chain $M$ is called *basic cylinder set* if there is a prefix $s_0, \alpha_1, \ldots \alpha_n, s_n$ such that $X$ contains exactly all runs of $M$ with that prefix. The probability meassure of a basic cylinder set with prefix $s_0, \alpha_1, \ldots \alpha_n, s_n$ is then

$$\prod_{i=0}^{n-1} P(s_i, \alpha_{i+1}, s_{i+1}).$$

If the set $X$ of runs of $M$ is not a basic cylinder set, its measure is determined as a sum of measures of maximal (w.r.t. inclusion) basic cylinder sets fully contained in $X$ [11].

In this paper we focus on the *quantitative model checking* of MDPs against properties specified in Linear temporal logic (LTL). Formulas of LTL are built over a set $AP$ of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective $X$ (next), and the binary temporal connective $U$ (until). LTL is interpreted over computations. A *computation* is a function $\pi : \omega \to 2^{AP}$, which assigns truth values to the elements of $AP$ at each time instant and as such it can be viewed as an infinite word over the alphabet $2^{AP}$. For an LTL formula $\varphi$, we denote by $\mathcal{L}(\varphi)$ the set of all computations satisfying $\varphi$.

A run of a Markov chain satisfies the formula $\varphi$, if the trajectory of the run is in $\mathcal{L}(\varphi)$. A Markov Chain $M$ satisfies the formula $\varphi$ with probability $p$, if the set of runs of $M$ satisfying the formula has the probability $p$. An MDP $M$ satisfies the formula $\varphi$ with the probability at least $p$ (at most $p$) if for every scheduler $D$, $M$ under $D$ satisfies the formula with the probability at least $p$ (at most $p$). The problem of quantitative model checking is to determine the minimal and/or maximal probability that an MDP satisfies a given property. Note that for the computation of the minimal and/or maximal probability that an MDP satisfies an $\omega$-regular property, it is sufficient to consider only history independent schedulers [12].

The goal of the *global* quantitative model checking is to calculate the minimal and/or maximal probability of the satisfaction of the property for every state $s$ of an MDP. The goal of the *local* quantitative model checking is, however, to determine the minimal and/or maximal probability of satisfaction of the property for the initial state only.

A Büchi automaton is a tuple $A = (\Sigma, Q, q_{init}, \delta, F)$, where $\Sigma$ is a finite alphabet, $Q$ is a finite set of states, $q_{init} \in Q$ is an initial state, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of *accepting states*. A run of $A$ over an infinite word $w = a_1 a_2 \ldots \in \Sigma^\omega$ is a sequence $q_0, q_1, \ldots$, where $q_0 = q_{init}$ and $(q_{i-1}, a_i, q_i) \in \delta$ for all $i \geq 1$. Let $\inf(\rho)$ denote the set of states that appear in the run $\rho$ infinitely often. A run $\rho$ is accepting iff $\inf(\rho) \cap F \neq \emptyset$. A state $s \in Q$ of a Büchi automaton $A$ is called *deterministic* if and only if for all $a \in \Sigma$ there is at most one $s' \in A$ such that $(s, a, s') \in \delta$. A Büchi automaton is *deterministic* *in the limit* if and only if all the accepting states and their descendants are deterministic [11].

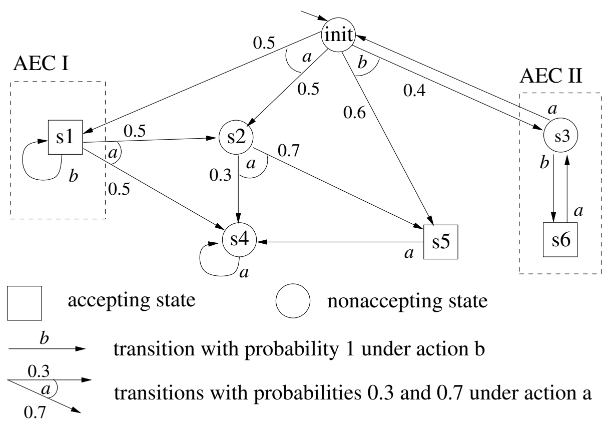
We use the automata based approach to probabilistic LTL model checking. Given an LTL formula $\varphi$, it is possible to build a Büchi automaton $A$ with $2^{\mathcal{O}(|\varphi|)}$ states such that $L(A) = \mathcal{L}(\varphi)$ [23]. Moreover, for any Büchi automaton $A$ with $n$ states a Büchi automaton $B$ with $2^{\mathcal{O}(n)}$ state such that $B$ is *deterministic in the limit* and $L(A) = L(B)$ can be built [11]. Similarly to model checking non-probabilistic systems, the model is synchronized with the automaton corresponding to the negation of the formula in the case we are interested in the minimal probability or with the automaton corresponding to the formula in the case we are interested in the maximal probability. However, unlike the non-probabilistic case, automata which are deterministic in the limit have to be used instead of non-deterministic Büchi automata.

Let $M = (S, Act, P, s_0, AP, L)$ be an MDP and let $A = (Q, 2^{AP}, q_0, \Delta, F)$ be a Büchi automaton. The synchronized product of $M$ and $A$ is an extended MDP $M \times A = (S \times Q, Act_{M\times A}, P_{M\times A}, init, AP, L_{M\times A}, Acc)$, where $Act_{M\times A}((s, p)) = Act(s)$, $P_{M\times A}((s, p), \alpha, (t, q)) = P(s, \alpha, t)$ if $(p, L(s), q) \in \delta$ or 0 otherwise, $init = (s_0, q_0)$, $L_{M\times A}((s, t)) = L(s)$, and $Acc = S \times F$ is the set of accepting states. Note that the synchronized product is not a regular MDP as it distinguishes between accepting and non-accepting states and may contain states without enabled actions.

In order to describe the algorithmic solution to the quantitative LTL model checking we often view an MDP or MDP synchronized with a Büchi automaton as a graph. Therefore, we recall some basic notions from the graph theory. A state $s'$ is *reachable* from a state $s$ in a set of states $R \subseteq S$, denoted as $s \leadsto_R^+ s'$ iff there is a sequence of states $s_0, s_1, \ldots, s_k \in R$ such that $s = s_0, s' = s_k$ and for all $0 \le i < k$ there is an action $\alpha \in Act(s_i)$ such that $P(s_i, \alpha, s_{i+1}) > 0$. A set of states $R$ is strongly connected if for all $r, r' \in R : r \leadsto_R^+ r'$ or $|R| = 1$. A *strongly connected component* (SCC) is a maximal strongly connected set of states. The graph of strongly connected components of $G$ is called the *quotient graph* of $G$. An SCC $C$ is *trivial* if $|C| = 1$. An SCC is *terminal* if it has no successors in the quotient graph. For every component $C$ let $Input(C) = \{c \in C \mid \text{there is an SCC } C' : \exists c' \in C' : \exists \alpha \in Act(c') : P(c', \alpha, c) > 0\}$ if $init \notin C$ otherwise $Input(C) = \{init\}$. Furthermore, for each nonterminal component $C$ we define $Output(C) = \{s \in S \setminus C \mid \exists c \in C : \exists \alpha \in Act(c) : P(c, \alpha, s) > 0\}$.

Given an MDP graph $G$, an *accepting end component* (AEC) is a maximal set $C$ of states of $G$ that forms (not necessary maximal) strongly connected component in $G$ such that $C \cap Acc \neq \emptyset$ and if there is an enabled action $\alpha$ in a state of the component, the component contains either all the $\alpha$-successors or none of them [12]. From [13,14] it follows that for any state $s$ of an MDP graph of $M \times A$ that belongs to an AEC, there exists a scheduler $D$ such that the probability measure of runs originating in $s$ and remaining in the AEC is 1 in $M$ under $D$.

Let $s$ be a state in the MDP product graph. We define the maximal probability $x_s$ of reaching an AEC from $s$ as follows. If $s$ belongs to an AEC, $x_s = 1$, if no


AEC I
AEC II
init
s1
s2
s3
s4
s5
s6
0.5
0.5
0.5
0.5
0.6
0.4
0.7
0.3
a
b
accepting state
nonaccepting state
transition with probability 1 under action b
transitions with probabilities 0.3 and 0.7 under action a

$$
\begin{aligned}
\min\ &:\ x_{init} \\
x_{init} &\geq 0.5 \cdot x_{s_1} + 0.5 \cdot x_{s_2} \\
x_{init} &\geq 0.4 \cdot x_{s_3} + 0.6 \cdot x_{s_5} \\
x_{s1} &\geq 1 \\
x_{s2} &\geq 0.3 \cdot x_{s_4} + 0.7 \cdot x_{s_5} \\
x_{s3} &\geq 1 \\
x_{s4} &\geq 1 \cdot x_{s_4} \\
x_{s5} &\geq 1 \cdot x_{s_4} \\
x_{s6} &\geq 1
\end{aligned}
$$

**Fig. 1.** MDP and its corresponding local linear programming problem

AEC is reachable from $s$, $x_s = 0$. For remaining cases the value of $x_s$ can be calculated by solving the linear programming problem with inequalities

$$x_s \geq \sum_{v \in S \times Q} P_{M \times A}(s, \alpha, v) \cdot x_v \quad \forall \alpha \in Act_{M \times A}(s)$$

minimizing the objective function $f = \sum_{u \in S \times Q} x_u$. For more details see [3,12]. Note that in the context of local model checking the objective function can be simplified to $f = x_{init}$. An example is given in Figure 1.

After the solution of the linear programming problem is found, $x_{init}$ contains the value of maximal probability an AEC is reached from the state *init*. If the MDP was synchronized with the automaton corresponding to the negation of a formula $\varphi$, the minimal probability the MDP satisfies the formula $\varphi$ is $1 - x_{init}$. If the MDP was synchronized with the automaton corresponding to a formula $\psi$ (without negation), the maximal probability the MDP satisfies $\psi$ is $x_{init}$.

## 2.2 Algorithm

The algorithm for finding all AECs was introduced in [11,12] and it was based on recursive decomposition of MDP graph into SCCs. Our approach employs a parallel adaptation of the algorithm of Bianco and de Alfaro (BdA) [7] that computes a set of states for which there exists a scheduler such that the maximal probability of reaching an AEC from the set is equal to 1. Clearly, this set can be used instead of the set of all AECs. Henceforward, the set is refered to as $AS$.

The algorithm maintains an *approximation set* of states that may belong to an AEC. The algorithm repeatedly refines the approximation set by locating and removing states that cannot belong to an AEC, we call this a *pruning step*. The algorithm for quantitative verification is obtained by a modification of BdA. As the final approximation set is $AS$, the linear programming problem is extended with inequalities $x_u \geq 1$ for all $x_u \in AS$. The overall scheme of how the algorithm proceeds is given as Algorithm 1.

**Algorithm 1.** Scheme of algorithm for local quantitative analysis

```
1: compute the set $AS$ using BdA algorithm
2: create the linear programming problem $LP$
3: compute the solution of $LP$
4: **return** $x_{init}$
```

## 3 Local Model Checking Techniques

In this section we introduce three optimization techniques that can significantly speed-up the verification process. We also propose a way how these techniques can be employed in a parallel environment, shared-memory multi-core architectures in our case. This is in particular very important in handling very large real-life systems in practice.

### 3.1 Minimal Subgraph Identification

The first of the proposed algorithmic modifications helps to reduce the size of the linear programming problem by pruning the MDP product $M \times A$ into the so called minimal subgraph.

The probability of a state depends on the probabilities of its successors. However, once we know that the probability of a state is 1, we do not need to know the exact probabilities of its successors. Also, the probability of a state is 0 if no state with probability 1 can be reached from it. Henceforward, we say that a state is *relevant* if it is on a path from an initial state to a state with probability 1 such that the path does not contain any other state with probability 1. The last state on the path is referred to as a *seed*. Relevant states define in a natural way a slice in the original MDP (see the example in Figure 2). We call this slice a *minimal subgraph*. The probability of the initial state is fully determined by the states in the minimal subgraph only.

With the minimal subgraph we associate the linear programming problem $mLP$ to be minimized in the following way. Let $init, s_0, s_1, \ldots, s_{r-1}, s_r$ be a path in the minimal subgraph from the initial state $init$ to a seed $s_r$. We add to $mLP$ the inequalities of form:

$$\begin{aligned} x_{init} &\geq \ldots + p_{s_0}x_{s_0} + \ldots \\ x_{s_0} &\geq \ldots + p_{s_1}x_{s_1} + \ldots \\ &\vdots \\ x_{s_{r-1}} &\geq \ldots + p_{s_r}x_{s_r} + \ldots \end{aligned}$$

It is not difficult to prove that pruning the original MDP graph into the minimal one does not have any influence on the solution of the linear programming problem.

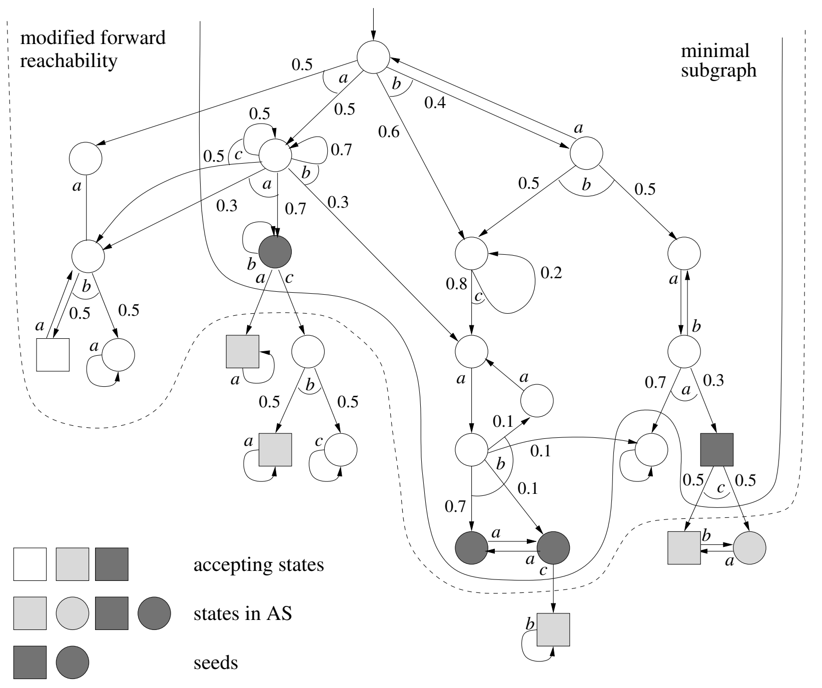


**Fig. 2.** Minimal subgraph identification

**Lemma 1.** *Let $M$ be a synchronous product of MDP and Büchi automaton, and MG its minimal subgraph. The solution of linear programming problem LP is equal to the solution of linear programming problem mLP.*

Having computed $AS$ we can identify relevant states, i.e. the minimal subgraph, as follows. First, we run a forward reachability from the initial state that does not explore states beyond a state from $AS$. States from $AS$ visited in this reachability are the seeds. Second, we run backward reachability from seeds to identify the minimal subgraph. All states visited by the backward reachability are relevant states. In this manner we omit states whose probability of reaching an AEC equals to zero. The result of applying both forward and backward reachability to obtain the minimal subgraph is illustrated in Figure 2.

## 3.2 Iterative Computation

Another practical technique is to decompose the given problem into simpler subtasks. In this way we have a good chance to end-up with a set of smaller linear programming problems that can be solved much faster.

The core idea is to decompose the minimal subgraph into SCCs, create the appropriate quotient graph, and then iteratively solve the linear programming problem by solving the subproblems given by the individual components in a bottom-up manner.

Some subtasks can be solved independently, which provides a basis for an effective parallel procedure as described in Subsection 3.4. Furthermore, we show in Subsection 3.3 that some subtasks can be solved without employing an external LP solver. Iterative computation can lead to a significant speed-up as compared to the computation of the entire LP problem (see Section 4).

Let us consider a minimal subgraph $MG$, its strongly connected components component $C$ is formed by all inequalities $x_s \geq \ldots$ from $mLP$ such that $s \in C$. The objective function of $LP_C$ minimizes the sum of variables $x_s$, $s \in Input(C)$, i.e. states with a predecessor outside the component, or the value $x_{init}$ in case $init \in C$.

The solution of $LP_C$ for each component $C$ depends only on $C$ itself and on the states in $Input(C_t)$ for each immediate successor component $C_t$ of $C$, as only variables corresponding to these states appear in the inequalities. This means, that for a terminal SCC $T$ we can find the solution of $LP_T$ directly. Once we have solutions for all successor components $C_t$ of $C$, we can substitute all the variables $x_s$, such that $s \notin C$, with already computed values. $LP_C$ does not depend on components $C_t$ any more and the solution of $LP_C$ can thus be computed. We call the SCC $C$ *solved* if $LP_C$ has been solved, i.e. the values $x_s$ for all $s \in Input(C)$ have been computed. An unsolved SCC $C$ is called *prepared* if for all $t \in Output(C)$ the state $t$ is in a solved SCC.

**Lemma 2.** *For each $s \in Input(C)$, the solution of $LP_C$ assigns to $x_s$ the value equal to the maximal probability that the set AS is reachable from $s$.*

**Corollary 1.** *For the component $C$ containing the initial state* init*, the solution of $LP_C$ assigns to $x_{init}$ the value equal to the maximal probability AEC is reachable from init.*

The pseudo-code of the iterative computation is described in Algorithm 2.

**Algorithm 2.** Iterative computation

```
Require: minimal subgraph $MG$
 1: decompose $MG$ into SCCs
 2: build the quotient graph of $MG$
 3: while there is an unsolved SCC do
 4:    compute the set $P$ of prepared SCCs
 5:    for all $C \in P$ do
 6:       create the linear programming problem $LP_C$
 7:       substitute for $x_s$ such that $s \in Output(C)$ in $LP_C$
 8:       compute the solution of $LP_C$
 9:       mark $C$ as solved
10:    end for
11: end while
12: return $x_{init}$
```

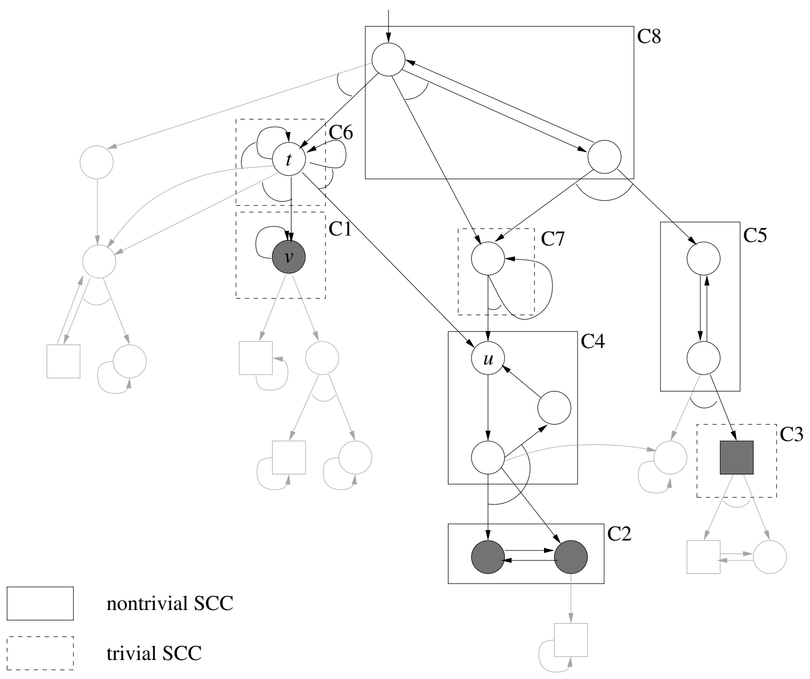


**Fig. 3.** SCC decomposition

Figure 3 shows the decomposition into strongly connected components. Components $C1, C2$ and $C3$ are terminal and thus prepared. After they are solved, components $C4$ and $C5$ become prepared. In the next iteration of the algorithm, components $C6$ and $C7$ are prepared and finally, after their solution, the component $C8$ containing the initial state is ready to be solved.

## 3.3 Trivial SCC

Let us suppose we perform the iterative computation on the minimal subgraph $MG$ as introduced in the previous Subsection, and let the next *prepared* SCC to be solved is a *trivial* strongly connected component $T = \{t\}$ with the corresponding linear programming subtask $LP_T$. In the following we show, that the linear programming subtask $LP_T$ can be solved without employing an external LP solver.

Let us firstly recall that due to deAlfaro [12] there is a *history independent* scheduler that yields the maximum value for the state $t$. We denote by $x_t^\alpha$ the probability of the state $t$ under the history independent scheduler choosing the action $\alpha \in Act(t)$ whenever the state $t$ is visited. Since it is sufficient to consider only history independent schedulers for the computation of the probability $x_t$ of the state $t$, it follows directly that

$$x_t = \max_{\alpha \in Act(t)} x_t^\alpha .$$

Lemma 3 says how to compute the value of $x_t^\alpha$. Before we state the lemma, we introduce the necessary notation. Suppose an action $\alpha$ to be executed. Let $u_0, u_1, \ldots, u_n$ be the $\alpha$-successors of $t$ that are outside the component $T$. Each $u_i$ is reached with the probability $P(t, \alpha, u_i)$ for $0 \leq i \leq n$. Let us denote these probabilities $p_{u_0}^\alpha, p_{u_1}^\alpha, \ldots, p_{u_n}^\alpha$, respectively. Since the states are outside the component $T$, the probability values for these states are already known and are refered to as $v_{u_0}, v_{u_1}, \ldots, v_{u_n}$. Futhermore, we denote the probability $P(t, \alpha, t)$ by $p_t^\alpha$.

**Lemma 3.** *Let $pv_u^\alpha = p_{u_0}^\alpha v_{u_0} + p_{u_1}^\alpha v_{u_1} + \ldots + p_{u_n}^\alpha v_{u_n}$. Then,*

$$x_t^\alpha = \begin{cases} \frac{pv_u^\alpha}{1-p_t^\alpha} & \text{if } p_t^\alpha \neq 1 \\ 0 & \text{otherwise.} \end{cases}$$

*Proof.* For a Markov chain the following holds:

$$\sum_{i=0}^{n} p_{u_i}^\alpha + p_t^\alpha \leq 1$$

Therefore, if $p_t^\alpha = 1$ then $p_{u_i}^\alpha = 0$ for all $0 \leq i \leq n$ and thus $x_t^\alpha = 0$. Otherwise

$$\begin{aligned} x_t^\alpha = & pv_u^\alpha + p_t^\alpha(pv_u^\alpha) + (p_t^\alpha)^2(pv_u^\alpha) + (p_t^\alpha)^3(pv_u^\alpha) + (p_t^\alpha)^4(pv_u^\alpha) + \ldots = \\ & pv_u^\alpha(1 + p_t^\alpha + (p_t^\alpha)^2 + (p_t^\alpha)^3 + (p_t^\alpha)^4 + \ldots) = pv_u^\alpha \frac{1}{1-p_t^\alpha} \end{aligned} \qquad \square$$

To compute the value of $x_t$ we enumerate the values $x_t^\alpha$ according to the previous Lemma and compute their maximum. For an example, we refer to Figures 2 and 3. The component $C6$ containing the state $t$ is a trivial one. After the components $C1$ and $C4$ are solved, we have $v_u = 0.9$, $u \in Input(C4)$ and $v_v = 1$, $v \in Input(C1)$. The value of $x_t$ can be now computed without employing the external LP solver. Altogether, there are three actions $a, b, c$ enabled in $t$ resulting in the following three cases:

$$x_t^a = \frac{pv_v^a}{1-p_t^a} = \frac{0.7 \cdot 1}{1} = 0.7 \quad x_t^b = \frac{pv_u^b}{1-p_t^b} = \frac{0.3 \cdot 0.9}{1-0.7} = 0.9 \quad x_t^c = \frac{0}{1-0.5} = 0$$

Finally, $x_t = \max(x_t^a, x_t^b, x_t^c) = 0.9$.

## 3.4 Parallelization

The improved algorithm, described as Algorithm 3, consists of several consecutive phases, each of them parallelized to a certain level.

Parallel version of BdA algorithm performs qualitative model checking and computes $AS$ as a basis for quantitative verification. The main idea builds on the topological sort for cycle detection – an algorithm that does not depend on DFS postorder and can be thus parallelized reasonably well. Minimal Subgraph Identification employs only one forward and one backward reachability and thus

**Algorithm 3.** Improved algorithm for local quantitative analysis

```
 1: compute the set $AS$ using **parallel** version of BdA algorithm
 2: compute the minimal subgraph $MG$ using **parallel** reachability
 3: decompose $MG$ into SCCs using **parallel** OBF algorithm
 4: build the quotient graph of $MG$ in **parallel**
 5: **while** there is an unsolved SCC **do**
 6:    compute the set $P$ of prepared SCCs
 7:    **for all** $C \in P$ **do**
 8:       **in parallel do**
 9:          **if** $C$ is trivial **then**
10:             compute the solution of $C$
11:          **else**
12:             create the linear programming problem $LP_C$
13:             substitute for $x_s$ such that $s \in Output(C)$ in $LP_C$
14:             compute the solution of $LP_C$
15:          **end if**
16:          mark $C$ as solved
17:       **end in parallel**
18:    **end for**
19: **end while**
20: **return** $x_{init}$
```

this phase is parallelized effectively. In order to parallelize SCC Decomposition, the implementation is based on recursive variant of OBF algorithm as described in [6]. Iterative Computation allows to solve prepared SCCs independently by parallel running threads. However, each component has to be solved by calling the external serial LP solver `lpsolve`. This last limitation could be eventually relaxed by a parallel LP solver (we were unfortunately not able to get access to a suitable free parallel solver).

## 4 Experimental Evaluation

We have implemented all the described algorithms and techniques in the tool called ProbDiVinE. The tool uses DiVinE Library [20] and a generally available LP solver `lpsolve`. We ran a set of experiments on machines equipped with Intel Xeon 5130 and AMD Opteron 885 processors allowing us to measure the performance of the tool when using 1 to 8 threads.

We have used five different experimental models of randomized protocols with properties yielding minimal probability other than 0 or 1:

- Cons – randomized consensus protocol [2]
- Crypts – randomized dining cryptographers [9]
- Leads – asynchronous leader election protocol [18]
- Phils – randomized dining philosophers [19]
- Stabi – randomized self-stabilizing protocol [17]

a)

| Model | time | | |
|---|---|---|---|
| | *no reduction* | *minimal subgraph* | *iterative solution* |
| Cons | 27.9 min | 11.9 min | 5.9 min |
| Crypts | >10 days | 17.8 min | 4.7 min |
| Leads | >10 days | >10 days | 3.7 min |
| Phils | >10 days | 2.5 days | 4.5 min |
| Stabi | >10 days | >10 days | 4.8 min |

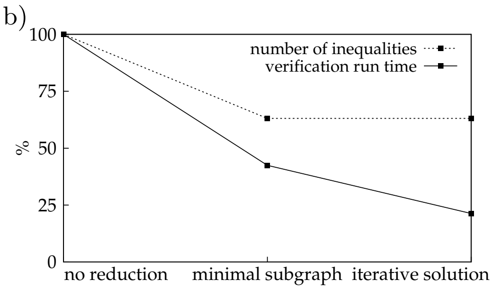
b)


**Fig. 4. a)** Runtimes for various models when no reduction is used, when only the minimal subgraph is considered, and when both the minimal subgraph and iterative processing is involved. **b)** Correlation between the number of inequalities and runtime.

Table 1 captures the size of the linear programming problem (the number of inequalities to be solved by an external LP solver). The column *whole graph* gives the size before applying any reduction, the column *reduced graph* gives the size when redundant inequalities were removed by pruning the graph into the minimal subgraph. The column *largest problem* gives the maximal size of a problem to be solved by an LP solver, when the original problem was decomposed into subproblems that were processed independently. Three of the models contain only trivial SCCs, thus the LP solver is not called at all and the size of the largest problem solved by LP solver is thus 0.

The table in Figure 4 demonstrates that the size and the structure of the problem plays a crucial role in the performance of the tool. The table gives overall run times corresponding to the used reduction techniques. The first column gives run time when no reduction technique is used (*no reduction*), the second one when redundant inequalities are removed (*minimal subgraph*), and the third one when the technique of iterative computation and trivial SC solving are applied on the minimal subgraph (*iterative solution*). A correlation between times in Figure 4 and sizes in Table 1 is observable. With decreasing number of inequalities the runtimes tend to speed-up dramatically. As for the speed-up, the most inconvenient case is when the graph is made of one large component. In such a case, the pruning and parallel processing cannot be done and the verification runtime is dominated by the single call to the LP solver.

**Table 1.** The size of LP problem with respect to used reduction techniques

| Model | # states | # inequalities for LP solver | | | % of the whole graph | |
|---|---|---|---|---|---|---|
| | | *whole graph* | *reduced graph* | *largest problem* | *reduced graph* | *largest problem* |
| Cons | 48 669 | 132 243 | 83 395 | 20 368 | 63.06 | 15.40 |
| Crypts | 2 951 903 | 8 954 217 | 108 045 | 0 | 1.21 | 0 |
| Leads | 2 995 379 | 8 800 096 | 5 678 656 | 0 | 64.5 | 0 |
| Phils | 5 967 065 | 14 740 726 | 1 623 722 | 246 | 11.0 | Almost 0 |
| Stabi | 4 061 570 | 6 897 480 | 5 983 080 | 0 | 86.7 | 0 |

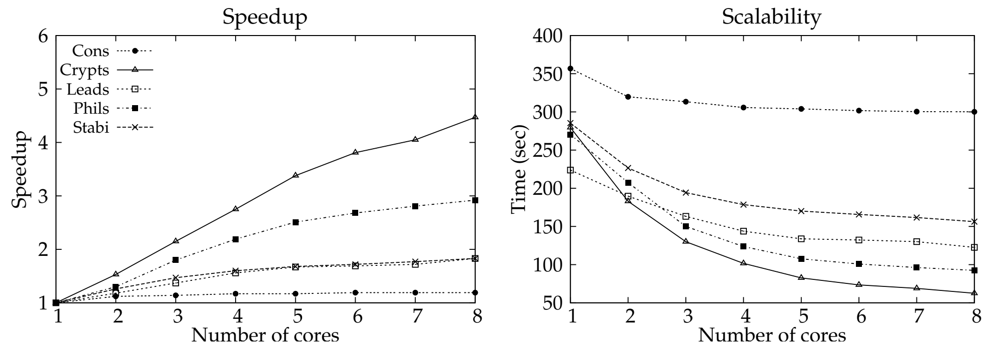


**Fig. 5.** Overall speed-up and scalability

In the case of *Cons* model the reduction techniques help less than in the other cases. However, run times decrease still significantly. Figure 4.a) gives runtimes of verification when various degree of improvement is used. Figure 4.b) depicts the relative decrease in the runtime and number of inequalities when various reduction techniques are involved. In particular, the number of inequalitites decreases to 63% of the original number, while the runtime decreases to 21% of the original time needed to perform the verification task.

Figure 5 reports on the overall speed-up and scalability of the verification process we achieved using our tool on various number of CPU cores. Poor scalability in case of *randomized consensus protocol* can be explained, because the time consumed by the sequential LP solver takes the major part of the runtime of the whole verification process.

Figure 6 aims on the qualitative analysis as a part of the whole verification process. The table in Figure 6 shows the ratio between runtimes of the qualitative analysis and the whole verification process. The graph in Figure 6 presents speed-up of qualitative analysis (the first phase of the algorithm). In comparison to the quantitative verification, the speed-up is much better due to the fact, that the whole verification process contains phases where parallelization does not help

| Model | *qualitative analysis* | *whole analysis* | *% qualit of whole* |
|---|---|---|---|
| Cons | 30.53 | 358.15 | 8.52 |
| Crypts | 275.96 | 279.47 | 98.75 |
| Leads | 68.28 | 223.76 | 30.51 |
| Phils | 180.6 | 270.46 | 66.77 |
| Stabi | 67.36 | 285.34 | 23.61 |

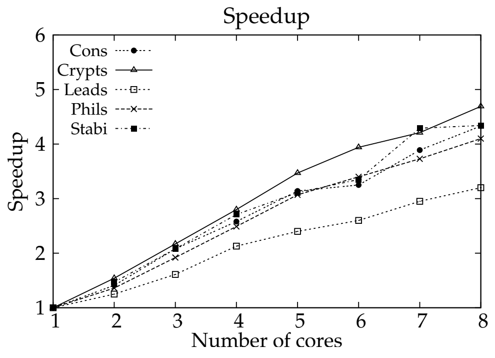


**Fig. 6.** Ratio between runtimes of the qualitative analysis and the whole verification process (table on the left). Speed-up of qualitative analysis only (on the right).

much as they require frequent synchronization. We can observe, that the bigger the part the qualitative analysis forms in the whole verification process, the better the overall scalability is. In case of *dining cryptographers protocol* model, the qualitative verification forms 98.75 % of the whole verification process and all the LPs are solved without using external LP solver. Therefore the scalability and speed-up are the best over all the examples.

All in all, we claim that our approach is quite successful as overall runtimes tend to decrease as more CPU cores are used.

The structure of a graph is a crucial aspect affecting the runtime of the verification process. For instance the *Crypts* model contains approximately the same number of states as *Leads*, but runtimes of qualitative verification differ a lot. On the other hand, runtime of *Leads* is comparable to *Stabi*, but their number of states and speedup differ.

## 5 Conclusion

As probabilistic systems gain popularity and are coming into wider use, the need for formal verification and analysis methods, techniques and tools capable of handling these systems become more critical. The theory and algorithms for formal verification of probabilistic systems have been around for some time. However, it is the existence of a good and efficient formal verification tool that makes the theory valid from the practitioner's point of view.

In this paper we presented several techniques that allow to build competitive enumerative model checking tool for quantitative analysis of linear temporal properties over finite state probabilistic systems. In particular, we showed how to involve parallelism and employ locality to increase the performance of such a tool. We also showed that the costly call to the linear programming solver can be either replaced with multiple successive calls for smaller problems, or avoided at all. Using this approach we achieved order-of-magnitude reduction in runtime of verification in many cases.

## References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous-time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
2. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms 15(1), 441–460 (1990)
3. Baier, C.: On the Algorithmic Verification of Probabilistic Systems. Habilitation Thesis, Universität Mannheim (1998)
4. Baier, C., Größer, M., Ciesinski, F.: Partial Order Reduction for Probabilistic Systems. In: 1st International Conference on Quantitative Evaluation of Systems (QEST 2004), pp. 230–239. IEEE Computer Society, Los Alamitos (2004)
5. Barnat, J., Brim, L., Černá, I., Češka, M., Tůmová, J.: ProbDiVinE-MC: Multi-Core LTL Model Checker for Probabilistic Systems. In: Proceedings of QEST 2008, Tool Paper. IEEE, Los Alamitos (2008) (to appear), `http://anna.fi.muni.cz/probdivine`

6. Barnat, J., Chaloupka, J., van de Pol, J.: Improved Distributed Algorithms for SCC Decomposition. Electron. Notes Theor. Comput. Sci. 198(1), 63–77 (2008)
7. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
8. Jeannet, B., de Argenio, P., Larsen, K.G.: RAPTURE: A tool for verifying Markov Decision Processes. In: Proc. Tools Day / CONCUR 2002. Tech. Rep. FIMU-RS-2002-05. MU Brno, pp. 84–98 (2002)
9. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
10. Ciesinski, F., Baier, C.: LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems. In: Proc. of QEST 2006, pp. 131–132. IEEE Computer Society, Los Alamitos (2006)
11. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
12. de Alfaro, L.: Formal Verification of Stochastic Systems. PhD thesis, Stanford University, Department of Computer Science (1997)
13. Derman, C.: Finite State Markovian Decision Processes. Academic Press, Inc., Orlando (1970)
14. Doob, J.L.: Measure theory. Springer, Heidelberg (1994)
15. Hansson, H., Jonsson, B.: A Framework for Reasoning about Time and Reliability. In: IEEE Real-Time Systems Symposium, pp. 102–111 (1989)
16. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
17. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizating mutual exclusion. In: Proc. ACM Symposium on Principles of Distributed Computing, pp. 119–131 (1990)
18. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation 88(1) (1990)
19. Lehmann, D., Rabin, M.: On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In: Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL 1981), pp. 133–138 (1981)
20. ProbDiVinE homepage (2008), `http://anna.fi.muni.cz/probdivine`
21. Puterman, M.L.: Markov Decision Processes-Discrete Stochastic Dynamic Programming. John Wiley &Sons, New York (1994)
22. Vardi, M.Y.: Probabilistic linear-time model checking: an overview of the automata-theoretic approach. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 265–276. Springer, Heidelberg (1999)
23. Vardi, M.Y., Wolper, P.: Reasoning about infinite computation paths. In: Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan, pp. 185–194 (1983)

2009 Sixth International Conference on Quantitative Evaluation of Systems

# Quantitative Model Checking of Systems with Degradation

Jiří Barnat, Ivana Černá, Jana Tůmová
*Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic*
{*xbarnat,cerna, xtumova*}*@fi.muni.cz*

***Abstract*—In this paper we describe a rather specialized quality of a system – the degradation. We demonstrate systems that naturally incorporate degradation phenomenon and we show how these systems can be verified by adapting the standard automata-based approach to LTL model checking. We introduce Büchi Automata with Degradation Constraints (BADCs) to specify the desired properties of systems with degradation and we describe how these can be used for verification. A major obstacle in the verification process is that the synchronous product of the system and the Büchi automaton may be infinite, which we deal with by introducing a normal form of the Büchi automata and normalizing procedure. We also show that the newly introduced formalism can be used to distinguish MDPs indistinguishable by any LTL, PCTL or even PCTL* formula.**

## I. Introduction

In order to reduce project design costs or to fit the tight time-to-market schedule, numerous software tools including formal verification ones are used in software and hardware development process. Quantitative properties of systems being developed are an inseparable part of the specifications in many cases. As a result, specialized software tools were designed and are publicly available to help system designers analyze various quantitative aspects of systems. For example, tools such as PRISM [1], LiQuor [2] or ProbDiVinE-MC [3] are used to design and analyze systems with probabilistic actions, tools such as UPPAAL [4] or KRONOS [5] are used to verify timing constraints of real-time systems, MRMC [6] tool analyzes Markov rewards, etc.

In this paper we introduce a rather specialized quality of a system – the degradation. The degradation phenomenon is quite common for objects that are subjects to physics laws. For example, we can measure the degradation of electric charge in some electronic devices, degradation of power or quality of a transmitted signal in broadcasting network, etc. However, the phenomenon is not bound to physical objects only and is present in many other kinds of systems including software ones. For example, a database index degrades with every database update, memory consistency degrades every time an allocation or deallocation of a memory block occurs (memory fragmentation), etc.

To model systems with degradation we use the following approach. Let us assume that an attribute of the model is subject to the degradation. The idea of the degradation is to express the consistency level (or quality) of the attribute using a real number. If the attribute is in perfect shape the associated number equals to one, if the consistency is degraded to 75%, the number equals to 0.75, etc. Since we do not admit negative consistency or consistency better than 100%, the number associated with the attribute is always a number between zero and one.

The level of degradation is manipulated by performing system actions. Every action of the system may either further degrade the attribute, or it may leave it as it is. Henceforward, we assume that the amount of degradation caused by an action of a system is associated with the action and is given as a real number again between zero and one. So, if the current level of degradation is $l$ and the degradation associated with an action is $d$, the new level of degradation will be $d \cdot l$ after the action is executed.

To our best knowledge, there are no appropriate formalisms developed to properly deal with the degradation aspects of a system. So far, the possibilities to handle the degradation might have been twofold. The first approach would involve using a standard model checker, e.g. SPIN [7]. We can introduce a floating point variable to keep the amount of degradation and describe how the degradation evolves by explicit manipulation with the variable. The second approach could be to use a formalism such as Markov Decision Processes (MDPs) to express the degradation phenomenon by means of probability. Unfortunately, neither of the approaches is suitable for modeling the real degradation phenomenon in more complex systems. In particular, both approaches lacks the general possibility to verify linear properties of runs of the system under consideration. For example, the property that system designers might be interested in is a repeated *response-with-limited-degradation*, such as: whenever A happens, B happens before the degradation of A drops below certain level. This property cannot be verified using the first approach as a fresh degradation variable needs to be introduced every time A happens. This would require a finite but unbounded number of degradation variables to be introduced in the system description, which is rather problematic regarding the restrictions of the standard model checker input languages. The other approach is unsuitable as well. MDPs require that a state of the system evolves into its immediate descendants in such a way that the sum of degradations distributed among the descendants equals

This work has been supported in part by the Czech Science Foundation grants No. 201/09/P497 and 201/09/1389.

978-0-7695-3808-2/09 $25.00 © 2009 IEEE
DOI 10.1109/QEST.2009.24

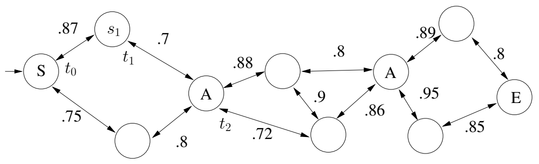


Figure 1. Signal coverage map.

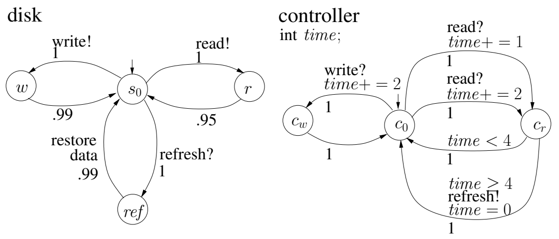


Figure 2. Magnetic disk example.

to one for a given action. This is quite restrictive and also unrealistic for many systems. For the same reasons, a system-wide fixed degradation constant, as suggested in [8], is inappropriate.

In this paper we demonstrate systems that naturally incorporate degradation phenomenon. We introduce quantitative linear properties that relate to systems with degradation and define Büchi Automata with Degradation Constraints as the standard formalism to express the desired degradation properties. We adapt the standard automata-based approach to LTL model checking to perform model checking procedure for quantitative linear properties over systems with degradation. The problem with the adaption is that the product automaton to be analyzed may be infinite. To avoid this, we suggest to transform the property Büchi automaton into the so called normal form which than guarantees finiteness of the product automaton. A separate section of the paper relates systems with degradation to MDPs. We demonstrate that expressive power of our specification formalism differs from that of PLTL, PCTL, or PCTL*.

## II. Systems with Degradation

### *Example I: Signal Coverage Problem*

Let us suppose, we want to get some signal from a start point $S$ to an end point $E$. Unfortunately, the points are too far from each other, so the signal cannot reach the destination without unrepairable signal degradation. A possible solution to the problem is to build relays in between $S$ and $E$ that restore the quality of the signal while the signal is still fully re-constructible. Furthermore, let us assume we have a map of possible places where a relay may be built including pairwise signal degradation values as illustrated in Figure 1. For the sake of simplicity, let us assume the signal goes through these places. Using a system with degradation, we can easily check, whether the signal reaches the target point in proper shape if the relays are built at the A-points. Another example of a degradation property might be to check whether some of the A-points are redundant.

### *Example II: Magnetic Disk*

A common problem that must be dealt with in a firmware of a storage device is a periodical refreshment of data being kept. There are numerous reasons for it, but for the sake of simplicity, let us just suppose that the data integrity are degraded by certain amount, let us say 5%, with every read operation. On the other hand every write or refresh operation restores the integrity of data to 99%. To be on a safe side, the producers of the storage device would like to guarantee that any piece of data is refreshed before its integrity drops below a certain level, let us say 85%. However, the device cannot simply refresh data after every read operation as this would lead to an unacceptable level of power consumption. Therefore, the data are refreshed periodically on a time basis. Note that the read operations may take various amount of time depending on the position of a reading head and the location from where the data are read, which we model using a non-deterministic choice. To answer the question whether the device meets the producers' requirements, we model the device and the controller as depicted in Figure 2. We can verify that no read action is performed if the data degradation is below 85% and refresh actions are performed only if the data degradation is below 90%.

### *Transition Systems with Degradation*

Informally, systems with degradation are systems that involve an attribute whose quality degrades (e.g. the data integrity in the magnetic disk example). We formalize such systems as *Transition Systems with Degradation* (TSDs). Unlike the standard transition systems, every transition is associated with a degradation constant in a TSD. A degradation constant is a rational number from interval $(0, 1]$. The constants may differ for individual transitions in the system. Note that the formal definition of a TSD contains no specification of the attribute that degrades, it only captures how much it degrades along each transition.

A *transition system with degradation* is a tuple $\mathcal{M} = (S, Act, \longrightarrow, S_{init}, AP, \mathcal{L})$, where

- $S$ is a finite, nonempty set of states,
- $Act$ is a finite, nonempty set of actions,
- $\longrightarrow \subseteq S \times Act \times (0, 1] \times S$ is a transition relation,
- $S_{init} \subseteq S$ is a set of initial states,
- $AP$ is a set of atomic propositions,
- $\mathcal{L} : S \rightarrow 2^{AP}$ is a labeling function; $\mathcal{L}(s)$ denotes the set of atomic propositions that are true in state $s$.

Instead of $(s_1, a, d, s_2) \in \longrightarrow$ we write $s_1 \xrightarrow{a,d} s_2$. A transition $s_1 \xrightarrow{a,d} s_2$ represents that the model can move from state $s_1$ to the state $s_2$ by a (nondeterministic) choice of action

$a$. The degradation constant $d$ associated with the transition gives the fraction to which the quality is degraded if the transition is executed. So if the level of degradation at state $s_1$ is let us say $l$ and the action is executed, the level of degradation at state $s_2$ will be $l \cdot d$.

A *path* in a TSD $\mathcal{M} = (S, Act, T, S_{init}, AP, \mathcal{L})$ is an infinite sequence $\pi = s_0 t_0 s_1 t_1 \ldots$ where $s_i \in S$ and $t_i = (s_i, a_i, d_i, s_{i+1}) \in T$ for all $i \geq 0$. A *trajectory* corresponding to the path $\pi = s_0 t_0 s_1 t_1 \ldots$ is given by the projection of $\pi$ to the state labels, $trajectory(\pi) = \mathcal{L}(s_0)\mathcal{L}(s_1)\ldots$. A *trace* corresponding to the path $\pi = s_0 t_0 s_1 t_1 \ldots$ is given by the projection of $\pi$ to the state labels and degradation rates, $trace(\pi) = (\mathcal{L}(s_0), d_0)\ (\mathcal{L}(s_1), d_1)\ \ldots$. Let $Traces(\mathcal{M})$ denote the set of all traces of paths in $\mathcal{M}$.

For instance, let us consider the example in Figure 1 and its path $St_0 s_1 t_1 A t_2 \ldots$ with the trace $(S, 0.87), (s_1, 0.7), (A, 0.72), \ldots$. The signal degradation between $S$ and $A$ is $0.87 \cdot 0.7 = 0.609$. This means the quality of the signal in $A$ will be $60.9\%$ of the quality in $S$.

## III. Quantitative Linear Properties and Büchi Automata with Degradation Constraints

One way to express a desired behavior of a system is to give restrictions on individual runs of the system, i.e. paths in its model. Properties specified by path restrictions are called linear and are defined on trajectories, i.e. sequences of atomic propositions holding true along a path. However, for systems with degradation, we might be interested not only in sequences of atomic propositions, but also in quantitative aspects, the amount of quality degradation in particular. Formally, we want to analyze traces rather than trajectories.

Consider a TSD $\mathcal{M} = (S, Act, \rightarrow, S_{init}, AP, \mathcal{L})$ and a path $\pi = s_0 t_0 s_1 t_1 s_2 t_2 \ldots$ in $\mathcal{M}$. The *amount of degradation along* $\pi$ *between states* $s_i$ *and* $s_j$, $i \leq j$, is defined as

$$D_i^j = \prod_{k=i}^{j-1} d_k.$$

In case $i = j$ the amount of degradation is equal to 1.

*Quantitative linear properties* are linear properties involving constraints on trajectories. These are expressed by specifying boundaries on the amount of degradation along a path between two states. Let us recall the signal coverage example. The question whether the signal reaches the target point in a proper shape is an example of quantitative linear property. In other words, we ask whether there exists a path from the sender to the receiver along which the amount of degradation of the signal does not drop below a given bound provided the signal is fully reconstructed in every relay (A-points). Another interesting quantitative linear question might be whether there are redundant relays on the way. A relay is redundant if the signal can reach properly its destination without being refreshed at the relay.

Regarding the magnetic disk example the question whether a piece of data is read when its degradation is below 85% or a piece of data is refreshed when its integrity is not below 90% is an example of quantitative linear property as well.

### *Büchi Automata with Degradation Constraints*

To express the quantitative linear properties of systems with degradation we introduce a modification of Büchi automata, the so called *Büchi Automata with Degradation Constraints* (BADC). The standard automata are enriched with a set of bounded variables allowing us to express the amount of degradation.

Let $D$ be a finite set of *degradation variables* ranging over the rational numbers in between $(0, 1]$. A *degradation constraint over* $D$ is of form

$$\varphi ::= x \bowtie d \mid \varphi \wedge \varphi,$$

where $\bowtie \in \{<, \leq, >, \geq\}$, $x \in D$, and $d$ is a rational number in $(0, 1]$. Note that degradation constraints exclude disjunction as it can be expressed using two different transitions of a BADC. $DC(D)$ denotes the set of degradation constraints over $D$. A *degradation valuation* is a function $\nu : D \rightarrow (0, 1]$. The set of all possible degradation valuations is $Eval(D)$.

A *Büchi Automaton with Degradation Constraints (BADC)* is a tuple $\mathcal{A} = (L, \Sigma, D, T, l_{init}, F)$, where

- $L$ is a finite nonempty set of states (locations),
- $\Sigma$ is a finite alphabet,
- $D$ is a finite set of degradation variables,
- $T \subseteq L \times \Sigma \times DC(D) \times 2^D \times L$ is a set of transitions,
- $l_{init} \in L$ is an initial location,
- $F \subseteq L$ is a finite set of locations (Büchi accepting condition).

A 5-tuple $t = (l, \alpha, \varphi, R, l') \in T$ represents the transition from location $l$ to $l'$ labeled with $\alpha$ that is enabled if constraint $\varphi$ is satisfied. $R$ is a set of degradation variables which are reset to 1 when executing the transition. For the transition $t = (l, \alpha, \varphi, R, l')$ we denote $label(t) = \alpha$, $constraint(t) = \varphi$ and $reset(t) = R$.

A *path* in a BADC $\mathcal{A} = (L, \Sigma, D, T, l_{init}, F)$ originating at location $l_0$ (or simply from $l_0$) is an infinite sequence of locations and transitions $\pi = l_0 t_0 l_1 t_1 \ldots$, where $l_i \in L$ and $t_i = (l_i, \alpha, \varphi, R, l_{i+1}) \in T$ for all $i \geq 0$.

A finite *path* from $l_0$ to $l_n$ is a finite prefix $\pi_{l_0}^{l_n} = l_0 t_0 l_1 \ldots l_{n-1} t_{n-1} l_n$ of a path from $l_0$. A finite path $\pi_{l_0}^{l_n}$ is *simple* if $\forall 0 \leq i, j \leq n-1, i \neq j$ implies $t_i \neq t_j$. A simple path $\pi_{l_0}^{l_n}$ forms an *elementary cycle* if $l_0 = l_n$ and $\forall 0 \leq i, j \leq n-1, i \neq j$ implies $l_i \neq l_j$.

The semantics of a BADC $\mathcal{A} = (L, \Sigma, D, T, l_{init}, F)$ is given by an infinite labeled transition system $\mathcal{M}_{\mathcal{A}} = (S, \Sigma', \rightarrow, S_{init})$, where

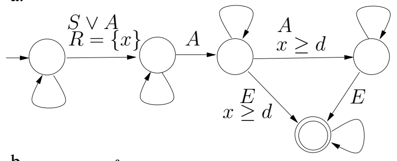
- $S = L \times Eval(D)$
- $\Sigma' = \Sigma \times (0, 1]$

- $\rightarrow \subseteq S \times \Sigma' \times S$
  $(l_1, \nu_1) \xrightarrow{\alpha, d} (l_2, \nu_2)$ whenever there is a transition $(l_1, \alpha, \varphi, R, l_2) \in T$ such that
  - $\nu_1 \models \varphi$
  - $\nu_2(x) = \begin{cases} d, & \text{if } x \in R \\ \nu_1(x) \cdot d & \text{otherwise} \end{cases}$
- $S_{init} = \{(l_{init}, \nu_{init}) \mid \nu_{init}(x) = 1 \text{ for all } x \in D\}$

A run for a word $\sigma = (\alpha_0, d_0)(\alpha_1, d_1) \ldots \in (\Sigma \times (0, 1])^\omega$ is an infinite sequence $\rho = (l_0, \nu_0)(l_1, \nu_1) \ldots$ such that $(l_0, \nu_0) \in S_{init}$ and $(l_i, \nu_i) \xrightarrow{\alpha_i, d_i} (l_{i+1}, \nu_{i+1})$ for all $i \geq 0$. A run $\rho = (l_0, \nu_0)(l_1, \nu_1) \ldots$ is accepting if $l_i \in F$ for infinitely many indices $i$. $L_\omega(\mathcal{A}) = \{\sigma \in (\Sigma \times (0, 1])^\omega \mid$ there exists an accepting run for $\sigma$ in $\mathcal{A}\}$.

Figures 3a and 3.b depict the "redundant A-point" quantitative linear property for the signal coverage example and the property of the magnetic disc example, respectively.

a.



b.

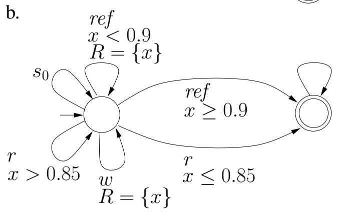


Figure 3. Quantitative properties of Sender/Receiver example (a) and Magnetic disc example (b).

## IV. Model Checking Algorithm

Model checking is a technique that for a given finite state model and a temporal property decides whether the model satisfies the property. In our case we are given a TSD model of a system with degradation and a BADC automaton specifying prohibited quantitative linear behaviors. In this section we develop an algorithm deciding whether a given TSD model exhibits a forbidden behavior. Our model checking algorithm follows the automata-based approach to LTL model checking [9]. First, we define a product automaton and prove that this automaton accepts exactly the intersection of the BADC language and the language of TSD traces. Next, we demonstrate that checking non-emptiness of the product automaton is equivalent to finding an accepting cycle in the product automaton graph and can be tested effectively by a number of known techniques like the Nested Depth First Search [10] or OWCTY [11].

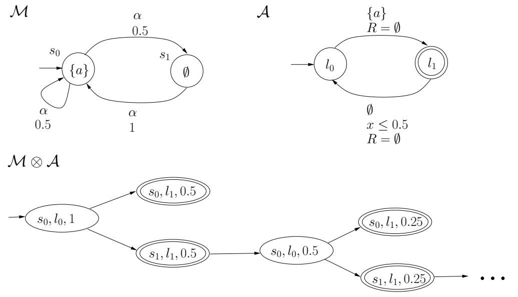


Figure 4. Infinite product.

### Product Automaton

*Product automaton* of a TSD $\mathcal{M} = (S, Act, \rightarrow, S_{init}, AP, \mathcal{L})$ and a BADC $\mathcal{A} = (L, 2^{AP}, D, T, l_{init}, F)$ is an automaton $\mathcal{M} \otimes \mathcal{A} = (Q, Act, \delta, Q_{init}, Q_F)$, where

- $Q = S \times L \times Eval(D)$
- $\delta : Q \times Act \rightarrow 2^Q$
  $(s', l', \nu') \in \delta((s, l, \nu), a)$ whenever
  - $\exists\ m = (s, a, d, s') \in \rightarrow$
  - $\exists\ t = (l, \mathcal{L}(s), \varphi, R, l') \in T$, such that $\nu \models \varphi$ and $\forall\ x \in D$ :

$$\nu'(x) = \begin{cases} d & \text{if } x \in R \\ \nu(x) \cdot d & \text{otherwise} \end{cases}$$

- $Q_{init} = \{(s_{init}, l_{init}, \nu_{init}) \mid s_{init} \in S_{init}$, and $\nu_{init}(x) = 1$ for all $x \in D\}$
- $Q_F = \{(s, l, \nu) \mid l \in F\}$

The product automaton $\mathcal{M} \otimes \mathcal{A}$ can be viewed as an oriented graph $G_{\mathcal{M} \otimes \mathcal{A}} = (Q, E)$. Vertices of $G_{\mathcal{M} \otimes \mathcal{A}}$ are the states of the product automaton and there is an edge from the vertex $(s, l, \nu)$ to the vertex $(s', l', \nu')$ if $\exists a \in Act : (s', l', \nu') \in \delta((s, l, \nu), a)$. Accepting cycle in the product automaton graph $G_{\mathcal{M} \otimes \mathcal{A}}$ is a cycle containing an accepting state. Henceforward, we consider only the subgraph of $G_{\mathcal{M} \otimes \mathcal{A}}$ reachable from the set of initial vertices $Q_{init}$, i.e. whenever we mention the product automaton graph, we implicitly mean its reachable subgraph.

We say that a product automaton $\mathcal{M} \otimes \mathcal{A}$ is finite if its graph is finite.

*Lemma 4.1:* If a product automaton $\mathcal{M} \otimes \mathcal{A}$ is finite then there is an accepting run of $\mathcal{M} \otimes \mathcal{A}$ if and only if the graph $G_{\mathcal{M} \otimes \mathcal{A}}$ contains an accepting cycle. [12]

The main obstacle in the verification process is that the product automaton graph may be infinite. An example of such a situation is depicted in Figure 4. Here, the infinity is caused by decreasing value of the variable $x$ always meeting the constraint $x \leq 0.5$. The key observation allowing for model checking of BADC properties of systems with degradation is that for a special type of BADC automata, the so called *normalized* BADC, it is guaranteed that the product graph is finite. In what follows we give the definition of a normalized BADC and prove that the product automaton of a TSD and a normalized BADC is finite. In the next section we provide an algorithm which transforms any BADC to an equivalent normalized BADC.

Let us consider a BADC $\mathcal{A} = (L, \Sigma, D, T, l_{init}, F)$. A degradation variable $x \in D$ is *in normal form* (or normalized, for short) in $\mathcal{A}$ if for each elementary cycle

$$\pi_{l_0}^{l_0} = l_0 t_0 l_1 t_1 \dots l_{n-1} t_{n-1} l_0$$

from $l_0$ to $l_0$ in $\mathcal{A}$ there is a transition $t_i, 0 \leq i \leq n-1$, such that at least one of the following two conditions holds:

- $constraint(t_i) = x \bowtie d$ or $x \bowtie d \wedge \psi$, where $d \in (0, 1]$, $\bowtie \in \{\geq, >\}$, and $\psi \in DC(D)$
- $x \in reset(t_i)$

$\mathcal{A}$ is *in normal form* (or normalized, for short) if each degradation variable $x \in D$ is in normal form in $\mathcal{A}$.

*Lemma 4.2:* The product automaton of a TSD $\mathcal{M} = (S, Act, \rightarrow, S_{init}, AP, \mathcal{L})$ and a normalized BADC $\mathcal{A} = (L, 2^{AP}, D, T, l_{init}, F)$ is finite.

*Proof:* To prove the finiteness of the graph $G_{\mathcal{M}\otimes\mathcal{A}}$ we have to demonstrate the finiteness of its set of states $Q \subseteq S \times L \times Eval(D)$. As $S$ and $L$ are both finite (from the definition of TSD and BADC) it is enough to prove that every constraint variable $x \in D$ attains only finitely many different values in $G_{\mathcal{M}\otimes\mathcal{A}}$.

Let $\rho = (s_0, l_0, \nu_0), (s_1, l_1, \nu_1) \dots (s_k, l_k, \nu_k)$ be a finite path such that the degradation variable $x$ is reset only in states $(s_0, l_0, \nu_0)$ and $(s_k, l_k, \nu_k)$. Formally, every edge $(s_i, l_i, \nu_i) \rightarrow (s_{i+1}, l_{i+1}, \nu_{i+1})$ of $\rho$ can be projected to the corresponding transition $m_i = (s_i, a_i, d_i, s_{i+1})$ of $\mathcal{M}$ and the transition $t_i = (l_i, \mathcal{L}(s_i), \varphi_i, R_i, l_{i+1})$ of $\mathcal{A}$. The variable is reset in a state $(s_i, l_i, \nu_i)$ iff $x \in R_i$. The initial value of $x$ on $\rho$ is $d_0$ and along the path is changed to $\prod_{i=0}^{1} d_i$, $\prod_{i=0}^{2} d_i$, $\dots$, $\prod_{i=0}^{k-1} d_i$, $d_k$. This sequence of $x$-values is non-increasing (with the possible exception of the last value $d_k$). We are to prove that there is a bound $B$ (depending only on $\mathcal{M}$ and $\mathcal{A}$) such that the value of $x$ is decreased on $\rho$ at most $B$ times. The existence of the bound $B$, together with the fact that there are only finitely many different degradation constants $d$ in transitions of $\mathcal{M}$, assure that $x$ attains only a finite number of different values along a path in $G_{\mathcal{M}\otimes\mathcal{A}}$.

We define constants $C_{\mathcal{M}}$, $C_{\mathcal{A}}$, and $L_{\mathcal{A}}$ distinguishing extremal values in $\mathcal{M}$ and $\mathcal{A}$. For the BADC $\mathcal{A}$ we define $C_{\mathcal{A}}$ as the minimal value such that there is a transition $t$ with $constraint(t) = x \bowtie C_{\mathcal{A}}$ or $x \bowtie C_{\mathcal{A}} \wedge \psi$, where $\bowtie \in \{\geq, >\}, d \in (0, 1]$ and $\psi \in DC(D)$. For the TSD $\mathcal{M}$ we define $C_{\mathcal{M}}$ as the minimal number such that the product of any $C_{\mathcal{M}}$ degradation constants $d$ from transitions of $\mathcal{M}$ is less than $C_{\mathcal{A}}$. $L_{\mathcal{A}}$ is the length of the longest elementary cycle in $\mathcal{A}$.

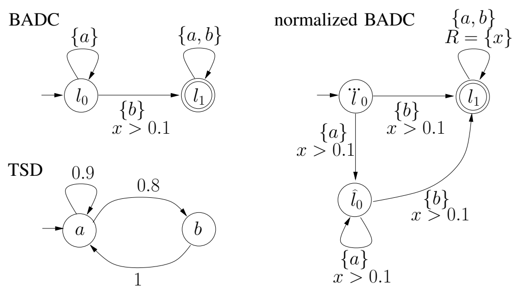


Figure 5. BADC, normalized BADC and TSD.

Let us suppose the value of $x$ is decreased on $\rho$ more than $C_{\mathcal{M}} + L_{\mathcal{A}}$ times. After the first $C_{\mathcal{M}}$ decreases the value of $x$ is less than $C_{\mathcal{A}}$. The length of the suffix $\rho'$ of $\rho$, starting in the state where the value of $x$ decreased below $C_{\mathcal{A}}$ for the first time, is greater than $L_{\mathcal{A}}$. The variable $x$ is normalized in $\mathcal{A}$ and thus there is a transition $(s_i, l_i, \nu_i) \rightarrow (s_{i+1}, l_{i+1}, \nu_{i+1})$ of $\rho'$ such that $constraint(t_i) = x \bowtie d$ or $x \bowtie d \wedge \psi$, where $\bowtie \in \{\geq, >\}, d \in (0, 1]$ and $\psi \in DC(D)$. However, this constraint cannot be satisfied as the value $\nu_i(x) < C_{\mathcal{A}} \leq d$. This contradicts the assumption about path $\rho$. ■

*Lemma 4.3:* $L_\omega(\mathcal{A}) \cap Traces(\mathcal{M}) \neq \emptyset \iff G_{\mathcal{M}\otimes\mathcal{A}}$ contains an accepting cycle.

*Proof:* $\Leftarrow$ Let $G_{\mathcal{M}\otimes\mathcal{A}}$ contain an accepting cycle. Then there is an infinite path $\rho = (s_0, l_0, \nu_0), (s_1, l_1, \nu_1), \dots$ in $G_{\mathcal{M}\otimes\mathcal{A}}$ with infinitely many accepting states. The projection of $\rho$ to the states of the corresponding TSD $\mathcal{M}$ gives the path $\pi = s_0 t_0 s_1 t_1 s_2 t_2 \dots$ with $trace(\pi) = (L(s_0), d_0))\ (L(s_1), d_1)), \dots$ . The projection $(l_0, \nu_0)(l_1, \nu_1)(l_2, \nu_2) \dots$ forms an accepting run of $\mathcal{A}$ for the $trace(\pi)$. Thus $L_\omega(\mathcal{A}) \cap Traces(\mathcal{M}) \neq \emptyset$.

$\Rightarrow$ Let $\sigma = (a_0, p_0)(a_1, p_1) \dots \in L_\omega(\mathcal{A}) \cap Traces(\mathcal{M})$. Then there is a path $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \dots$ in $\mathcal{M}$ with $trace(\pi) = \sigma = (L(s_0), d_0)\ (L(s_1), d_1) \dots$ where $t_i = (s_i, a_i, d_i, s_{i+1}) \in T$ for all $i \geq 0$. Let $\rho' = (l_0, \nu_0)(l_1, \nu_1)(l_2, \nu_2) \dots$ be an accepting run for $\sigma = trace(\pi)$ in $\mathcal{A}$. Then $l_0 = l_{init}, \nu_0(c) = 1$ for $c \in C, (l_i, \nu_i) \xrightarrow{L(s_i), p_i}_{\mathcal{A}} (l_{i+1}, \nu_{i+1})$ for all $i \geq 0$. Furthermore, there are infinitely many indices $i$ with $l_i \in F$.

Synchronizing the path $\pi = s_0 a_0 s_1 a_1 \dots$ and the run $\rho' = (l_0, \nu_0)(l_1, \nu_1) \dots$ we obtain a run $\rho = (s_0, l_0, \nu_0)(s_1, l_1, \nu_1)(s_2, l_2, \nu_2) \dots$ in the product $\mathcal{M} \otimes \mathcal{A}$ with infinitely many indices $i$, such that $(s_i, l_i, \nu_i) \in Q_F$, i.e. $G_{\mathcal{M}\otimes\mathcal{A}}$ contains an accepting cycle. ■

The number of states of the product is $\mathcal{O}(|S| \cdot |L| \cdot \Pi_{d \in D_N} \log_{step} min(d))$, where $|S|$ is the number of states of a TSD, $|L|$ is the number of locations in an BADC before normalization, $D_N$ is the set of degradation variables after normalization, $step$ is the maximal degradation constant different from 1 occurring in the TSD, and $min(d)$ is the minimal threshold connected with degradation variable $d$ occurring in the BADC.

An optional way to construct the product automaton without normalization is to modify the procedure of construction of the product automaton as follows. As soon as the value of a degradation variable drops below the minimal threshold occurring in the BADC, the value is tagged with a special flag denoting *below minimal threshold* and it is not manipulated in succeeding states anymore. This approach leads to a finite product automaton with $\mathcal{O}(|S| \cdot |L| \cdot (\log_{step} min)^{|D|})$ states, where $|S|$, $|L|$, and $step$ are as in the previous case, $|D|$ is the number of degradation variables, and $min$ is the overall minimal threshold occurring in the BADC.

The reason, why we have introduced normalization is that it helps to rapidly reduce the size of the product automaton in many cases. It is basically a heuristic to minimize the number of different values each degradation variable may get. Figure 5 illustrates an original BADC, its normalized form and a transition system. The product of the original BADC and the TSD has 239 states, whereas in the case of the normalized BADC the product has only 46 states. The normalization procedure adds resets of variables whenever it is possible. See, e.g., the self-loop on state $l_1$.

## V. Normalization of BADC

In this section, we describe how to transform a BADC into an equivalent BADC in the normal form.

Let us say that a degradation variable $x$ is *bounded* in degradation constraint $\varphi \in DC(D)$ if $\varphi = x \bowtie d$ or $\varphi = x \bowtie d \wedge \psi$, $\psi \in DC(D)$. More precisely, $x$ is *n-bounded* in constraint $\varphi \in DC(D)$ if $\varphi = x \bowtie_1 d_1 \wedge \ldots \wedge x \bowtie_n d_n$ or $\varphi = x \bowtie_1 d_1 \wedge \ldots \wedge x \bowtie_n d_n \wedge \psi$, where $x$ is not bounded in $\psi$. $x$ is ($n$-)bounded in transition $t$ if $x$ is ($n$-)bounded in $constraint(t)$.

**Algorithm 1** Normalization of BADC

```
Input:   BADC $\mathcal{A}^0 = (L^0, \Sigma, D^0, T^0, l^0_{init}, F^0)$
Output:  BADC $\mathcal{A} = (L, \Sigma, D, T, l_{init}, F)$ in normal form
 1: $\mathcal{A} := \mathcal{A}^0$
 2: OneConstraintOnVariable
 3: $Done := \emptyset$
 4: while $Done \neq D$ do
 5:     pick $x \in D \setminus Done$
 6:     $(L_R, L_P, L_x)$ := ResetWherePossible($x$)
 7:     SplitIntoLayers($x$)
 8:     $Done := Done \cup \{x\}$
 9:     Assert: Each $x \in Done$ is normalized in $\mathcal{A}$
10: end while
11: Assert: $\mathcal{A}$ is in normal form
12: Assert: $L_\omega(\mathcal{A}^0) = L_\omega(\mathcal{A})$
```

The transformation algorithm (see Algorithm 1) works in several stages (for an illustrative example see [13]). In the initial stage (see Procedure OneConstraintOnVariable), the given BADC is transformed into a BADC in which every degradation variable $x$ is bounded by at most one inequality $x \bowtie d_x$. This is accomplished by introducing new degradation variables into the BADC. In the next stage, we iteratively pick a degradation variable $x \in DC$ and transform the BADC so that $x$ becomes normalized while preserving the normal form of the already processed degradation variables.

Normalization of the degradation variable $x$ involves two procedures. The first procedure (see Procedure ResetWherePossible) identifies those transitions where the variable $x$ can be safely reset. To this end it computes the set $\Pi$ of all simple paths $\pi$ satisfying three conditions: $\pi$ starts in the initial location or in a location immediately after reset of $x$, no reset of $x$ occurs along $\pi$, and $\pi$ ends in a location from which there is a transition with a bound on $x$. Now we can split all the transitions into two disjoint sets: those which occur on a path from $\Pi$ (the set $T_P$) and those which

**Algorithm 2** Procedure OneConstraintOnVariable

```
 1: for all $x \in D^0$ do
 2:     $\mathcal{A}' := \mathcal{A}$
 3:     $T_x := \{t_i \in T \mid x$ is bounded in $t_i\}$
 4:     for all $t_i \in T_x$ do
 5:         $m_i := n$ such that $x$ is $n$-bounded in $t_i$
 6:         $D := (D \setminus \{x\}) \cup \{x_{i1} \ldots, x_{im_i} \mid x_{i1}, \ldots, x_{im_i}$ are new,
            unique degradation variables$\}$
 7:         replace $constraint(t_i) = x \bowtie_1 d_1 \wedge \ldots \wedge x \bowtie_{m_i} d_{m_i} \wedge \varphi$
            with $x_{i1} \bowtie_1 d_1 \wedge \ldots \wedge x_{im_i} \bowtie_{m_i} d_{m_i} \wedge \varphi$
 8:     end for
 9:     for all $t \in T$ do
10:         if $x \in reset(t)$ then
11:             $reset(t) := (reset(t) \setminus \{x\}) \cup \{x_{i1}, \ldots, x_{im_i} \mid t_i \in$
                $T_x\}$
12:         end if
13:     end for
14:     Assert: $L_\omega(\mathcal{A}') = L_\omega(\mathcal{A})$
15: end for
16: Assert: $\forall x \in D : \exists! t \in T$ with bounded $x$
17: Assert: $\forall x \in D : \exists! t \in T$ with 1-bounded $x$
18: Assert: $L_\omega(\mathcal{A}^0) = L_\omega(\mathcal{A})$
```

**Algorithm 3** Procedure ResetWherePossible($x$)

```
 1: $T_x := \{t_x \in T \mid x$ is bounded in $t_x\}$
 2: $L_x := \{l_x \in L \mid \exists$ transition $t_x \in T_x$ from location $l_x\}$
 3: $\Pi := \{\pi \mid \pi$ is a simple path $l_0 t_0 l_1 \ldots l_n t_n l_x$ in $A$, $l_0 = l_{init}$
    or $\exists$ transition $t \in T$ to $l_0$ with $x \in reset(t)$, $\forall\ 0 \leq i \leq n$ :
    $x \notin reset(t_i)$, and $l_x \in L_x\}$
 4: $L_R := \{l_0 \mid \exists\ \pi \in \Pi$ originating at $l_0\}$
 5: $L_P := \{l_i, l_x \mid \exists\ \pi = l_0 t_0 \ldots l_n t_n l_x \in \Pi, 0 \leq i \leq n\}$
 6: $T_P := \{t_i \mid \exists\ \pi = l_0 t_0 \ldots l_n t_n l_x \in \Pi, 0 \leq i \leq n\}$
 7: $T_N := T \setminus T_P$
 8: for all $t \in T_N$ do
 9:     $reset(t) := reset(t) \cup \{x\}$
10: end for
11: return $(L_R, L_P, L_x)$
12: Assert: $L_\omega(\mathcal{A}^0) = L_\omega(\mathcal{A})$
```

**Algorithm 4** **Procedure** SPLITINTOLAYERS($x$)

```
1: Let $x \bowtie d$ be the constraint on $x$
2: $lb(x) := \bowtie \in \{<, \leq\}\ ?\ x \bowtie d : \neg(x \bowtie d)$
3: $L := L \cup \{\widehat{l}, \check{l} \mid l \in L_P, \widehat{l}, \check{l}$ are new, unique locations$\} \cup \{\dddot{l} \mid l \in L_R, \dddot{l}$ is a new, unique location$\}$
4: $l_{init} := \dddot{l}_{init}$ if $l_{init} \in L_R$
5: $F := (F \setminus L_P) \cup \{\widehat{l}, \check{l} \mid l \in L_P \cap F\} \cup \{\dddot{l} \mid l \in L_R \cap F\}$
6: for all $t = (l_1, a, \varphi, R, l_2) \notin T_x$ do
7:    case $l_1 \notin L_P$, $x \in R$, and $l_2 \in L_R$: replace $t$ with $(l_1, a, \varphi, R, \dddot{l}_2)$
8:    case $l_1 \in L_P$, $x \in R$, and $l_2 \in L_R$: replace $t$ with $(\widehat{l_1}, a, \varphi, R, \dddot{l}_2)$, and $(\check{l}, a, \varphi, R, \dddot{l}_R)$
9:    case $l_1 \in L_R$, $x \in R$, and $l_2 \in L_R$: add transition $(\dddot{l}_1, a, \varphi, R, \dddot{l}_2)$
10:   case $l_1 \in L_P$, $x \notin R$, and $l_2 \in L_P$: replace $t$ with $(\widehat{l_1}, a, \varphi \wedge \neg lb(x), R, \widehat{l_2})$, $(\widehat{l_1}, a, \varphi \wedge lb(x), R \cup \{x\}, \check{l}_2)$, and $(\check{l}_1, a, \varphi, R \cup \{x\}, \check{l}_2)$
11:   case $l_1 \in L_R$, $x \notin R$, and $l_2 \in L_P$: add transitions $(\dddot{l}_1, a, \varphi \wedge \neg lb(x), R, \widehat{l_2})$, and $(\dddot{l}_1, a, \varphi \wedge lb(x), R \cup \{x\}, \check{l}_2)$
12:   case $l_1 \in L_P$, and $l_2 \notin L_P$: replace $t$ with $(\widehat{l_1}, a, \varphi \wedge, R \cup \{x\}, l_2)$, and $(\check{l}_1, a, \varphi, R \cup \{x\}, l_2)$
13:   case $l_1 \in L_R$, and $l_2 \notin L_P$: add $(\dddot{l}_1, a, \varphi, R \cup \{x\}, l_2)$
14: end for
15: for all $t_x = (l_x, a, \varphi, R, l_2) \in T_x$ do
16:   if $\bowtie \in \{<, \leq\}$ then
17:      $\bar{l}_2 := l_2 \notin L_P\ ?\ l_2\ :\ (x \in R\ ?\ \dddot{l}_2\ :\ \check{l}_2)$
18:      replace $t_x$ with $(\widehat{l_x}, a, \varphi \wedge lb(x), R \cup \{x\}, \bar{l}_2)$, $(\check{l}_x, a, \varphi, R \cup \{x\}, \bar{l}_2)$, and if $l_x \in L_R$ add $(\dddot{l}_x, a, \varphi \wedge lb(x), R \cup \{x\}, \bar{l})$
19:   else
20:      $\bar{l}_2 := l_2 \notin L_P\ ?\ l_2\ :\ (x \in R\ ?\ \dddot{l}_2\ :\ \widehat{l_2})$
21:      replace $t_x$ with $(\widehat{l_x}, a, \varphi \wedge \neg lb(x), R, \bar{l}_2)$, and if $l_x \in L_R$ add $(\dddot{l}_x, a, \varphi \wedge \neg lb(x), R, \bar{l})$
22:   end if
23: end for
24: Assert: $L_\omega(\mathcal{A}^0) = L_\omega(\mathcal{A})$
```

do not (the set $T_N$). We can reset the variable $x$ on the transitions from $T_N$ without changing the language of the BADC. Simultaneously, three other sets of locations, namely $L_R$, $L_P$, and $L_x$, are computed. $L_R$ is the set of locations in which a path $\pi \in \Pi$ originates. $L_P$ are locations occurring along a path $\pi \in \Pi$, and finally $L_x$ are locations in which a path $\pi \in \Pi$ ends. Note that $L_R, L_x \subseteq L_P$.

Procedure SPLITINTOLAYERS finishes the normalization of the variable $x$. It manipulates the rest of the transitions that may cause that $x$ is not normalized, namely those from the set $T_P$. The modification of the BADC is a bit more involved here and requires a replication of locations. Each replica of the location bears a specific information about the actual value of $x$. We replace each location $l \in L_P$ with two new locations $\check{l}$ and $\widehat{l}$. Moreover, if $l \in L_R$, we introduce a new location $\dddot{l}$. The information associated with the replicas is intuitively characterized as follows:

- $\dddot{l}$-locations: Whenever the location $l \in L_R$ is entered via a transition with reset of $x$ from location $k$ in the original BADC, the location $\dddot{l}$ is entered in the transformed one from location $k$ if $k \notin L_P$ or from any replica of $k$ if $k \in L_P$. The value $\nu(x)$ is the same in $\dddot{l}$ and in the corresponding $l$ in the original BADC.
- $\check{l}$-locations: Let $x \bowtie d_x$ be the only degradation constraint which bounds $x$ in $\mathcal{A}$. Let us define a lower bound $lb(x)$ as $lb(x) = x \bowtie d_x$ if $\bowtie \in \{<, \leq\}$ and $lb(x) = \neg(x \bowtie d_x)$ otherwise. Whenever the location $l \in L_P$ is entered from a location $k$ in which $\nu(x) \models lb(x)$ via a transition without reset of $x$ in the original BADC, the location $\check{l}$ is entered in the transformed one from any replica of $k$ (necessarily, $k \in L_P$). Due to the monotonicity of degradation, starting from the state $k$ the value of $x$ remains less or less-or-equal than $d_x$ until a reset of $x$. Therefore, we do not need to keep the value $\nu(x)$ in $\check{l}$ the same as in $l$ (it suffices to know that $\nu(x)$ remains below $d_x$). Thus we can add reset of $x$ on each transition entering the $\check{l}$-location.
- $\widehat{l}$-locations: $\widehat{l}$-locations are dual to $\check{l}$-locations. Whenever the location $l \in L_P$ is entered from a location $k$ in which $\nu(x) \not\models lb(x)$ via a transition without reset of $x$ in the original BADC, the location $\widehat{l}$ is entered in the transformed one from $\widehat{k}$ and in case $k \in L_R$ also from $\dddot{k}$. It cannot be entered from $\check{k}$ as we know that $\nu(x) \models lb(x)$ in $\check{k}$. The value $\nu(x)$ is the same in $\widehat{l}$-location and in the corresponding $l$-location in the original BADC. Note that any transition leading to a $\widehat{l}$-location contains a bound of form $x > d_x$ or $x \geq d_x$.

Transitions entering $l$ are naturally replaced by transitions entering $\dddot{l}, \check{l}$ or $\widehat{l}$ keeping the above characteristics. Normal form is guaranteed by the fact that for every degradation variable $S$ every transition in the resulting BADC either resets the value of $x$ or contain a constraint of the form $x > d_x$ or $x \geq d_x$.

The correctness of the construction is proved and the complexity issues are examined in [13]. The algorithm complexity is $\mathcal{O}(2^{2n} \cdot (|L| + |T|))$, where $|L|$ is the number of locations in the input BADC, $|T|$ is the number of tran-

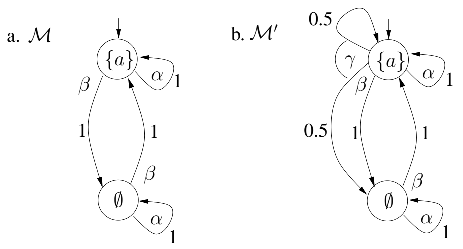


Figure 6. MDPs indistinguishable by any LTL, PCTL, or PCTL* formula.

sitions, and $n$ denotes the overall number of occurrences of all degradation variables in the BADC (i.e. $\Sigma_{d \in D} \Sigma_{t \in T} m$, where $D$ is the set of degradation variables, and $d$ is $m$-bounded in $t$).

## VI. Quantitative Linear Properties of Markov Decision Processes

This section raises the question about the parallel between the systems with degradation and the Markov decision processes (MDPs) [12], [14], [15] as well as about the relationship between probabilistic logic PLTL, PCTL, PCTL* and the quantitative linear properties formalized via BADCs. It is easy to see that an MDP is just a special case of a system with degradation. However, Büchi automata with degradation constraints can distinguish otherwise indistinguishable MDPs.

Current model checking of MDPs aims particularly on properties expressed in *LTL* (Linear Temporal Logic) [16], *PCTL* (Probabilistic Computation Tree Logic) [17] and *PCTL** [18]. The problem of quantitative LTL model checking of an MDP is to determine minimal and/or maximal probability (w.r.t. all possible schedulers) of a set of paths in the MDP that satisfy the LTL formula. PCTL and PCTL* verification gives an answer to the question whether a given MDP satisfies a PCTL (or PCTL*) state formula.

### *MDPs as Transition Systems with Degradation*

Let us consider a transition system with degradation $\mathcal{M} = (S, Act, \rightarrow, S_{init}, AP, \mathcal{L})$ and extend it with the following restrictions on the transition relation $\rightarrow$:

- for all $s_1, s_2 \in S$, $a \in Act$ there is at most one $d$ such that $(s_1, a, d, s_2) \in \rightarrow$
- for all $s_1 \in S$, $a \in Act : \sum_{(s_1,a,d,s_2) \in \rightarrow} d = 1$ or $0$.

We may think of the probability as a quality of the system that degrades in time. If probabilities are interpreted as degradations, the restricted transition systems with degradation are syntactically equivalent to MDPs.

### *MDPs and Temporal Properties*

In this subsection we show two MDPs which cannot be distinguished by any LTL, PCTL or even PCTL* formulas. First, let us consider the MDP $\mathcal{M} = (S = \{s, t\}, Act = \{\alpha, \beta\}, P, s, \{a\}, \mathcal{L})$ as illustrated in Figure 6.a.

We show that the minimal and the maximal probability of a set of paths originating at a particular state and satisfying a linear temporal property is always either 0 or 1.

*Observation 6.1:* Let $\eta$ be an arbitrary scheduler for $\mathcal{M}$. Then the Markov chain induced by $\eta$ is $\mathcal{M}_\eta = (S^+, P_\eta, s_{init}, AP, \mathcal{L}_\eta)$, where
$P_\eta(s_0 \dots s_n, s_0 \dots s_n s_{n+1}) = P(s_n, \eta(s_0 \dots s_n), s_{n+1})$

$$= \begin{cases} 1 & \text{if } s_n = s_{n+1} \text{ and } \eta(s_0 \dots s_n) = \alpha \text{ or} \\ & \text{if } s_n \neq s_{n+1} \text{ and } \eta(s_0 \dots s_n) = \beta \\ 0 & \text{otherwise} \end{cases}$$

and $\mathcal{L}_\eta(s_0 \dots s_n) = \mathcal{L}(s_n)$.

Note that for each state $s_0 \dots s_n$ in $\mathcal{M}_\eta$ there is exactly one state $s_0 \dots s_n s_{n+1}$ such that $P_\eta(s_0 \dots s_n, s_0 \dots s_n s_{n+1}) > 0$. In other words, there is exactly one path $\pi = (s_0)(s_0 s_1)(s_0 s_1 s_2) \dots$ in $\mathcal{M}_\eta$ originating at $s_0$. The set of all paths originating at $s_0$ in $\mathcal{M}_\eta$ is $\{\pi\}$ and its probability is 1. Similarly, there is exactly one path $\pi = (s_0 s_1)(s_0 s_1 s_2)(s_0 s_1 s_2 s_3) \dots$ in $\mathcal{M}_\eta$ originating at $s_0 s_1$ and the probability of the set of all paths $\{\pi\}$ originating at $s_0 s_1$ is equal to 1.

Let us consider the language $L$ of words over the alphabet $2^{\{a\}}$ representing a linear temporal property. For each symbol $\gamma \in 2^{\{a\}}$ we distinguish three possible cases:

1) there is no word in $L$ starting with $\gamma$,
2) $L$ contains all words starting with $\gamma$, or
3) $\exists\ \sigma_1 = \gamma(2^{\{a\}})^\omega \in L$ and $\exists\ \sigma_2 = \gamma(2^{\{a\}})^\omega \notin L$

To simplify the following discussion we denote by symbol $u$ the state $s$ of $\mathcal{M}$ in case of $\gamma = \{a\}$ and the state $t$ otherwise (i.e. if $\gamma = \emptyset$).

*Lemma 6.2:* Suppose there is no word in $L$ starting with the symbol $\gamma$. Then the minimal and the maximal probability of the set of paths of $\mathcal{M}$ originating at $u$ with trajectories in $L$ is 0.

*Proof:* It follows directly from the fact that there is no path $\pi = u\alpha_0 s_1 \alpha_1 s_2 \alpha_2 \dots$ in $\mathcal{M}$ with the trajectory $\mathcal{L}(u)\mathcal{L}(s_1)\mathcal{L}(s_2) \dots \in L$. ■

*Lemma 6.3:* Suppose $L$ contains all words starting with $\gamma$. Then the minimal and the maximal probability of the set of paths of $\mathcal{M}$ originating at $u$ with trajectories in $L$ is 1.

*Proof:* For each path $\pi = u\alpha_0 s_1 \alpha_1 s_2 \alpha_2 \dots$ in $\mathcal{M}$ originating at $u$ it holds that the corresponding trajectory $\mathcal{L}(u)\mathcal{L}(s_1)\mathcal{L}(s_2) \dots = \gamma\mathcal{L}(s_1)\mathcal{L}(s_2) \dots$ is present in $L$. Thus the probability of the set of paths originating at $u$ with trajectories in $L$ is 1 for any possible scheduler $\eta$ in $\mathcal{M}$. ■

*Lemma 6.4:* Let us suppose there are $\sigma_1 = \gamma(2^{\{a\}})^\omega \in L$ and $\sigma_2 = \gamma(2^{\{a\}})^\omega \notin L$. Then the maximal probability of the set of paths of $\mathcal{M}$ originating at $u$ with trajectories in $L$ is 1 and the minimal probability is 0.

| $L$ | probability of the set of paths with trajectories in $L$ | | | |
|---|---|---|---|---|
| | origin. at s | | origin. at t | |
| | min | max | min | max |
| $\{a\}(2^{\{a\}})^\omega \cap L = \emptyset$ | 0 | 0 | – | – |
| $\{a\}(2^{\{a\}})^\omega \subseteq L$ | 1 | 1 | – | – |
| $\{a\}(2^{\{a\}})^\omega \cap L \cap co-L \neq \emptyset$ | 0 | 1 | – | – |
| $\emptyset(2^{\{a\}})^\omega \cap L = \emptyset$ | – | – | 0 | 0 |
| $\emptyset(2^{\{a\}})^\omega \subseteq L$ | – | – | 1 | 1 |
| $\emptyset(2^{\{a\}})^\omega \cap L \cap co-L \neq \emptyset$ | – | – | 0 | 1 |

Table I
SUMMARY OF RESULTS, LEMMA 6.2 - 6.4

*Proof:* We define schedulers $\eta_1$ and $\eta_2$ for $\mathcal{M}$ such that the trajectory of the paths originating at $u$ in the induced Markov chain $\mathcal{M}_{\eta_1}$ and $\mathcal{M}_{\eta_2}$ are $\sigma_1$ and $\sigma_2$, respectively.

Let $\sigma_1 = \gamma A_1 A_2 \ldots$. The scheduler $\eta_1$ is defined by the prescription ($s_0 = u$)

$$\eta_1(s_0 \ldots s_n) = \begin{cases} \alpha & \text{if } \mathcal{L}(s_0 \ldots s_n) = A_{n+1} \\ \beta & \text{if } \mathcal{L}(s_0 \ldots s_n) \neq A_{n+1}. \end{cases}$$

The scheduler $\eta_1$ unambiguously determines the only path in $\mathcal{M}_{\eta_1}$ and the trajectory of this path is $\sigma_1$. This fact together with the Observation 6.1 implies that the maximal probability of the set of paths of $\mathcal{M}$ originating at $u$ with trajectories in $L$ is 1.

For the minimal probability and the scheduler $\eta_2$ the arguments are similar. ■

We summarize results given by Lemmas 6.2 - 6.4 in Table I. The symbol '–' indicates that we cannot say anything about the probability bound. Note that any language $L$ satisfies exactly one of the three cases given on the first three lines of the table and exactly one of the three cases given on the second three lines of the table. Therefore, given an arbitrary linear temporal property $L$, the minimal and the maximal probability for the system $M$ can be completely determined using just the table.

Let us now consider an MDP $\mathcal{M}'$ in Figure 6.b. Using similar arguments as for the MDP $\mathcal{M}$ we obtain the very same results about probability bounds for linear temporal properties for $\mathcal{M}'$, for the summary see Table I.

The minimal and the maximal probabilities of the set of paths originating at the initial states $s$ and $s'$ with trajectories in $L$ are the same for the MDP $\mathcal{M}$ and the MDP $\mathcal{M}'$, respectively. The same observation holds for the states $t$ and $t'$. Thus there is a one-to-one correspondence between the states $s$ and $s'$ and also between the states $t$ and $t'$. Therefore MDPs $\mathcal{M}$ and $\mathcal{M}'$ cannot be distinguished neither by qualitative verification nor by quantitative verification with any LTL formula.

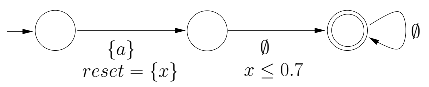


Figure 7. BADC distinguishing the two MDPs.

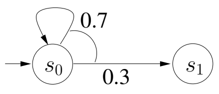


Figure 8. MDP demonstrating interesting BADC property.

Furthermore, if $\varphi$ is a CTL path formula then both the minimal and the maximal probability of the set of paths satisfying $\varphi$ is always either 0 or 1 for all the states both in $\mathcal{M}$ and $\mathcal{M}'$. Hence, for any PCTL or PCTL* formula $P_{\bowtie p}\varphi$ it holds that $\mathcal{M} \models P_{\bowtie p}\varphi \Leftrightarrow \mathcal{M}' \models P_{\bowtie p}\varphi$. As a result, the difference between $\mathcal{M}$ and $\mathcal{M}'$ cannot be captured by any PCTL or PCTL* formula.

### *MDPs and Quantitative Linear Properties*

Now we are to define a quantitative linear property which allows us to distinguish the Markov decision processes $\mathcal{M}$ and $\mathcal{M}'$. The property is specified by the BADC in Figure 7. The property captures the existence of a path with the trajectory $\{a\}\emptyset^\omega$ such that the amount of degradation (probability) between the state $s$ ($s'$) and the first next occurrence of the state $t$ ($t'$, respectively) is at most 0.7. This property is false for $\mathcal{M}$ (there is only one path with the trajectory $\{a\}\emptyset^\omega$ and the amount of degradation is 1), but is true for $\mathcal{M}'$ (there is a path where the amount of degradation is 0.5).

Using BADCs for expressing properties of MDPs brings us a new possibility to check for the presence of a specific path with a certain probability contribution. See for example the MDP as depicted in Figure 8. The probability of reaching $s_1$ from $s_0$ is 1 for all (there is only 1) schedulers. Every finite path from $s_0$ to $s_1$ (there are infinitely many of them) contribute to the resulting probability measure with some portion. With BADC approach we can, for example, verify that the mentioned portion exceeds $0.2$ for some paths, but is at most $0.3$ for all paths.

## VII. CONCLUSIONS

Degradation phenomenon as presented in this paper is important from two different points of view. First, it allows system designers to capture and analyze new kind of qualities of their systems, which itself is quite interesting. A second aspect is that the new degradation approach provides a new theoretical way to describe and analyze quantitative linear properties for probabilistic systems, such as MDPs. A limited-degradation-response property is a nice example of a property evaluated over a single run, hence a property that cannot be expressed in any formalism built upon some probability measures. Linear properties and LTL model checking in particular, are well established and used-in-practice formalism. A sort of linear property verification approach for probabilistic systems has been missing so far.

We stress that the degradation cannot be easily modeled using other well known formalisms. Many of other formalisms are either too restrictive to express and check a limited-degradation-response properties, e.g. the standard non-deterministic systems or MDPs. Other formalisms are so rich that a general model checking procedure is undecidable, which is the case of e.g. general hybrid systems, other formalism are simply focused to other quantitative aspects of systems like, e.g. real-time model checking, or model checking rewards.

A straightforward extension is to define a sort of extended linear temporal logic that would allow us to express the desired degradation properties as formulas. For example, the limited-degradation-response property could be stated in an LTL like formalism as follows: $G(A \implies F_{\leq 0.8}B))$. An inseparable part of this task is also to design a transformation procedure that would for a given formula produce the corresponding normalized BADC. Finally, let us mention that we have implemented a prototype model checker that is able to verify MDPs against properties given as normalized BADCs on top of our verification tool set DiVinE [19] allowing thus to employ parallel architectures to verify large-scale systems. The models to be verified by DiVinE model checker are given as networks of asynchronously communicating extended finite automata. For the purpose of verification of systems with degradation, we only extended individual automata with the possibility of specification of individual degradation constants.

## VIII. Acknowledgment

We thank anonymous referees for their kind recommendations to improve the paper.

## References

[1] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *Proc. of TACAS'06*, ser. LNCS, vol. 3920. Springer, 2006, pp. 441–444.

[2] F. Ciesinski and C. Baier, "LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems," in *Proc. of QEST'06*. IEEE Computer Society, 2006, pp. 131–132.

[3] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová, "Probdivine-mc: Multi-core ltl model checker for probabilistic systems," in *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–78.

[4] G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, and W. Yi, "UPPAAL - present and future," in *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[5] S. Yovine, "Kronos: A verification tool for real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 123–133, 1997.

[6] J.-P. Katoen, M. Khattri, and I. S. Zapreev, "A Markov reward model checker," in *Quantitative Evaluation of Systems (QEST)*. IEEE Computer Society, 2005, pp. 243–244.

[7] G. J. Holzmann, "The model checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997. [Online]. Available: citeseer.ist.psu.edu/holzmann97model.html

[8] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga, "Model checking discounted temporal properties," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 139–170, 2005.

[9] M. Vardi and P. Wolper, "Reasoning about infinite computation paths," *Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan*, pp. 185–194, 1983.

[10] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 275–288, 1992. [Online]. Available: citeseer.ist.psu.edu/courcoubetis92memoryefficient.html

[11] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Zijiang, "Is there a best symbolic cycle-detection algorithm," in *In Proc. Tools and Algorithms for Construction and Analysis of Systems, volume 2031 of LNCS*. Springer, 2001, pp. 420–434.

[12] C. Baier and J. P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[13] J. Barnat, I. Černá, and J. Tůmová, "Quantitative Model Checking of Systems with Degradation," Faculty of Informatics, Masaryk University, Tech. Rep. FIMU-RS-2009-04, 2009.

[14] C. Derman, *Finite State Markovian Decision Processes*. Orlando, FL, USA: Academic Press, Inc., 1970.

[15] M. L. Puterman, *Markov Decision Processes-Discrete Stochastic Dynamic Programming*. John Wiley &Sons, New York, 1994.

[16] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, 1977, pp. 46–57.

[17] H. Hansson and B. Jonsson, "A Framework for Reasoning about Time and Reliability," in *IEEE Real-Time Systems Symposium*, 1989, pp. 102–111.

[18] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "It usually works: The temporal logic of stochastic systems," in *Proceedings of the 7th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1995, pp. 155–165.

[19] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "DiVinE – A Tool for Distributed Verification (Tool Paper)," in *Computer Aided Verification*, ser. LNCS, vol. 4144/2006. Springer Berlin / Heidelberg, 2006, pp. 278–281.

# Chapter 6

# Tool Papers

1. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

2. J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

3. J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. *Electronic Proceedings in Theoretical Computer Science (PDMC 2009)*, 14:107–111, 2009.

4. J. Barnat, L. Brim, and P. Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.

5. J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 215–216. IEEE Computer Society, 2007.

6. J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Probdivine-mc: Multi-core ltl model checker for probabilistic systems. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 77–78, Washington, DC, USA, 2008. IEEE Computer Society.

# DiVinE – A Tool for Distributed Verification[⋆] (Tool Paper)

Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec,
Petr Ročkai, and Pavel Šimeček

Faculty of Informatics, Masaryk University, Brno, Czech Republic

**Abstract.** We present a tool for cluster-based LTL model-checking and reachability analysis. The tool incorporates several novel distributed-memory algorithms and provides a unique interface to use them. We describe the basic structure of the tool, discuss the main architecture decisions made, and briefly explain how the tool can be used.

## 1 Introduction

A few enumerative verification tools have been developed to support engineers in their verification needs. Despite significant improvements in model-checking techniques, their verification capabilities are in the case of real-life industrial models limited by the amount of data a *single* state-of-the-art computer is able to handle efficiently.

In recent years, extensive research has been conducted in parallel and distributed model-checking with the aim to push forward the frontiers of enumeratively verifiable systems [1,3,4,6,8]. Consequently, several distributed verification prototype tools emerged. The deployment and usage of a distributed tool is significantly more demanding compared to the sequential one. It assumes a cluster with properly installed message passing software and also some programming skills are required in the case the tool has to be compiled from its source codes. These are some of the reasons why distributed verification tools are used rarely, although their verification capabilities are undoubtedly bigger in comparison to the sequential tools.

The goal of Distributed Verification Environment project (DiVinE) is to provide an extensible framework to support distributed verification on clusters. DiVinE offers three means to achieve this goal: First, a library of common functions (DiVinE Library) on top of which various distributed verification algorithms can be implemented. Second, a collection of state-of-the-art distributed verification algorithms incorporated into a single software product (DiVinE Tool) which is as easy to install as most sequential tools. And third, a ready-to-use cluster for users of sequential tools in case they need to run experiments using DiVinE Tool without having access to their own cluster. In this paper we report on DiVinE Tool only.

⋆ This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

## 2 DiVinE Tool

DiVinE Tool is a parallel, distributed-memory enumerative model-checking tool for verification of concurrent systems. The tool employs aggregate power of network-interconnected workstations to verify systems whose verification is beyond capabilities of sequential tools.

DiVinE modelling language is rich enough to describe systems made of synchronous and asynchronous processes communicating via shared memory and buffered or unbuffered channels. System properties can be specified either directly in Linear Temporal Logic (LTL) or alternatively as processes describing undesired behaviour of systems under consideration (negative claim automata). Thanks to the DivSPIN project [2], DiVinE Tool is also capable of verifying models written in ProMeLa.

From the algorithmic point of view, the tool is quite unique. In automata-based approach to LTL model-checking, the verification problem is reduced to problem of accepting cycle detection in the graph of Büchi automaton. Two algorithms are typically used for solving the problem: Nested Depth-First Search algorithm and Tarjan's algorithm for decomposition of the graph into strongly connected components. Unfortunately, they both strongly rely on depth-first search postorder that is known to be difficult to be computed in parallel. Therefore, new, principally different, parallel algorithms for accepting cycle detection had to be designed. These are, namely, algorithm for cycle detection using additional dependency data structure, algorithm based on negative cycles, algorithms for forward and backward elimination of trivial and non-accepting strongly connected components, algorithm for cycle detection based on breadth-first search, and algorithm based on propagation of the value of maximal accepting predecessor(see [1] for an overview). Besides these, DiVinE Tool includes also an algorithm for distributed state space generation and an algorithm that performs sequential NestedDFS in a distributed-memory setting. More details on algorithms can be found on DiVinE project web pages [5].

DiVinE Tool can be deployed either as a complete software package to be installed on a separate Linux cluster or as a small Java application to access a pre-installed clusters. In the first case, basic Linux administrator skills are required to install the tool, but the user is in the full control of environment settings under which distributed algorithms are to be executed and can control the tool from a command line. In the second case, the tool can be used employing DiVinE pre-installed clusters and accessed remotely via a graphical user interface. The graphical user interface (GUI) requires properly installed Java Runtime Environment. Both versions are available on DiVinE project web page [5] together with a few models determined for initial acquaintance with the tool.

An important part of the DiVinE project is the maintenance of a *public* server together with a limited number of DiVinE dedicated clusters. For security reasons registered users are allowed to connect to DiVinE public server only. New users can be registered by following instructions given on DiVinE project web pages.

## 3 Interacting with DiVinE by Using GUI

The description of command line interface is beyond the scope of this paper. Therefore, we focus on controlling DiVinE Tool with GUI only. GUI is implemented as a client-server application where the server part is responsible for the control of the tool. This means the server maintains currently verified models, executes distributed algorithms, monitors cluster load, etc.

The client window is divided into three parts. In the main working area, models and properties are specified, and outputs of distributed algorithms are displayed. Another part of the client window has a tree-like structure and is used to browse currently loaded models including corresponding properties and verification results. The third part displays messages reporting changes in the status of running algorithms.

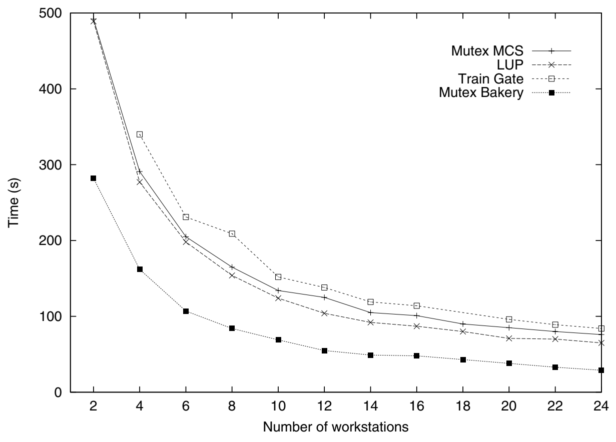
A new *verification project* is started by pressing the *New model* button. The system to be verified can be written directly into the main window or imported from a local file. Having specified a model of the system the user is expected to provide properties the system should meet using the button *Add property.* Besides distributed state space generation, the tool is capable of verifying full range of LTL formulae over state-based atomic propositions. Atomic propositions are specified using the keyword `#define`, e.g. `#define p x>3`, the formula is specified using the keyword `#property`, e.g. `#property FG(p)`. Property specification can also be imported from a local file. The pair model-property is called a *task.* User can assign several distributed algorithms to be run for a given task. The number of workstations to be used can be specified for every algorithm as well. Individual algorithms are initiated with the button *Execute.*

Each algorithm produces two different types of output that can be accessed with the client: the standard output and log files. While the standard output is used to report progress in the computation and final verification results, logs are used to generate multiple statistics to support the performance analysis. For each computer participating in the computation, the logged values include the amount of memory currently allocated by the algorithm, number of sent and received messages, time spent in user and kernel space, size of queue of unexplored states, etc. Client displays the last logged values with refresh rate around five seconds, which allows the user to monitor the status of the computation in almost real time.

All specified models, properties and verification results are stored on the server until they are explicitly removed. Therefore, the user can disconnect from the server, while initiated algorithms are still running, and reconnect later to collect the verification results. It is also possible to specify and initiate new tasks during computation of others. Hence, several tasks can be computed in parallel.

## 4 Conclusion

DiVinE is a tool for enumerative model checking of LTL properties on a cluster of workstations. We performed numerous experiments that clearly demonstrates the tool is capable to handle systems intractable by a single machine. E.g. for some classical verification problems the results on a cluster with 20 workstations were: Anderson's mutual exclusion problem – space required was 10GB of memory/verification took about 40 minutes, Dining Philosophers – 9GB/20 minutes, Leader Election – 17GB/46 minutes. For more examples see the tool web page. Another interesting performance characteristic is the scalability. The figure shows typical behaviour of algorithms with respect to the number of workstations involved.



## References

1. J. Barnat, L. Brim, and I. Černá. Distributed Analysis of Large Systems. In *Proc. of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 05)*, LNCS. Springer, 2006.
2. J. Barnat, V. Forejt, M. Leucker, and M. Weber. DivSPIN – A SPIN compatible distributed model checker. In *Proc. 4th International Workshop on Parallel and Distributed Methods in verifiCation*, pages 95–100, 2005.
3. G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proc. of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.
4. B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free $\mu$-calculus. In *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.
5. DiVinE project web page: http://anna.fi.muni.cz/divine/.
6. H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In *Proc. of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.
7. F. Holmen, M. Leucker, and M. Lindstrom. Uppdmc: A distributed model checker for fragments of the mu-calculus. *Electronic Notes in Theoretical Computer Science*, 128(3):91–105, 2005.
8. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

# DiVinE Multi-Core – A Parallel LTL Model-Checker[⋆]

Jiri Barnat, Lubos Brim, and Petr Ročkai[⋆⋆]

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

**Abstract.** We present a tool for parallel shared-memory enumerative LTL model-checking and reachability analysis. The tool is based on distributed-memory algorithms reimplemented specifically for multi-core and multi-cpu environments using shared memory. We show how the parallel algorithms allow the tool to exploit the power of contemporary hardware, which is based on increasing number of CPU cores in a single system, as opposed to increasing speed of a single CPU core.

## 1 Introduction

DiVinE Multi-Core has evolved from DiVinE Tool [2], sharing its input language and state space generator. As DiVinE Tool, it is an enumerative LTL model checker based on parallel fair cycle detection. The full source code can be obtained from [6] and compiled on a number of computer architectures.

The groundwork of tool design and algorithm choice has been laid down in [1]. We have crafted a tool from the ground up with shared-memory parallelism in mind. Due to natural choices of algorithms and memory organisation (the latter explored in more detail in [3]), the tool implementation closely resembles a distributed-memory one and may lend itself to extension to clusters of multi-core machines.

The primary motivation behind DiVinE Multi-Core has always been performance. Until recently, improvements in hardware architecture have been providing verification tools with performance increases mostly for free – without any need for implementational or algorithmic changes in the tools. However, this trend appears to be diminishing in favour of increasing parallelism in the system – which is nowadays much cheaper and easier to implement than it is to build computers with even faster sequential operation.

However, this architectural shift means that it is no longer possible to benefit from hardware progress, without introducing algorithmic changes to our tools. This is what DiVinE Multi-Core is striving for – providing algorithms able to exploit such parallel architectures and offering an implementation that can

[⋆] This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

[⋆⋆] This work has been partially supported by Red Hat, Inc.

be deployed in practical situations. The main challenging aspect of the design of a parallel application is to achieve practical scalability – a decrease in runtime with an increase in the number of involved CPU cores.

Other researchers have recognized this trend and multi-core algorithms have been added to at least to previously purely serial model checker `SPIN` [7]. Unfortunately, only a dual-core algorithm has been devised for full LTL model checking, limiting the multi-core capabilities to reachability analysis.

## 2 DiVinE-MC Algortihm

DiVinE Multi-Core is based on automata-theoretic approach to LTL model checking [9]. The input language allows for specification of processes in terms of extended finite automata and the verified system is then obtained as an asynchronous parallel composition of these processes. This system is in turn synchronously composed with a property process (negative claim automaton) obtained from the verified LTL formula through a Büchi automaton construction.

The resulting finite product automaton is then checked for presence of accepting cycles (fair cycles), indicating nonemptiness of its accepted language – which in turn indicates invalidity of the verified LTL property.

The algorithm employed for accepting cycle detection is OWCTY [5] augmented with a heuristic for on-the-fly cycle detection inspired by the MAP algorithm [4]. It is not the purpose of this tool paper to go into details of the algorithm, so for in-depth description, we refer the reader to the two cited papers.

The main idea behind the OWCTY algorithm is to use topological sort for cycle detection – an algorithm that does not depend on DFS postorder and can be thus parallelized reasonably well. Detection of cycles in this way is linear, but since we do accepting cycle detection, provisions for removing non-accepting cycles need to be added. This makes the algorithm quadratic in the worst case for general LTL properties, although for a significant subset of formulae (those that translate to weak Büchi automata) the algorithm runs in linear time in the size of the product automaton.

The MAP algorithm uses maximal accepting predecessors to identify accepting cycles in the product automaton. The main idea is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. Instead of expensively computing and storing all accepting predecessors for each accepting vertex (which would be sufficient to conclude the presence of an accepting cycle), the algorithm computes only a single representative accepting predecessor for each vertex – the maximal one in a suitable ordering. Clearly, if an accepting vertex is its own maximal accepting predecessor then it lies on an accepting cycle. This condition is used as the heuristic mentioned above. Note that the opposite direction does not hold in general. It can happen that the maximal accepting predecessor for an accepting vertex on a cycle does not lie on the cycle and the original MAP algorithm employs additional techniques to handle such a case.

```
process P_$1 {
byte j=0, k=0;
state NCS, CS, wait, q2, q3;
init NCS;
trans
 NCS -> wait { effect j = 1, active = $1, waiting[$1] = 1; },
 wait -> q2 { guard j < N;
              effect pos[$1] = j, active = $1; },
 q2 -> q3 { effect step[j-1] = $1, k = 0, active = $1; },
 q3 -> q3 { guard (k == $1 || pos[k]< j) && k < N;
            effect k = k+1, active = $1; },
 q3 -> wait { guard step[j-1] != $1 || k == N;
              effect j = j+1, active = $1; },
 wait -> CS { guard j == N;
              effect in_critical = in_critical+1,
                     active = $1, waiting[$1] = 0; },
 CS -> NCS { effect pos[$1] = 0, in_critical = in_critical-1,
                    active = $1; };
}
```

**Fig. 1.** An example of model specification: A single process participating in peterson mutual exclusion protocol (the $1 placeholder signifies the id of the process)

```
#define a_0 (active == 0)
#define a_1 (active == 1)

#define w_0 (waiting[0] == 1)
#define w_1 (waiting[1] == 1)

#define c_0 (P_0.CS)
#define c_1 (P_1.CS)

#property G(F(c_0)) && G(F(c_1))
#property ((GF(a_0 && w_0)) -> GF(c_0)) && ((GF(a_1 && w_1)) -> GF(c_1))
```

**Fig. 2.** Atomic propositions and LTL properties for the model in a `.ltl` file

If the heuristic fails, the OWCTY run will still detect accepting cycles if present. The heuristic does not interfere in any way when there are no accepting cycles – OWCTY will detect that condition by itself. The cost of the heuristic is a very slight increase in per-state memory usage, and a small runtime penalty in the first phase of the first iteration of OWCTY. Overall, it does not increase the time complexity compared to OWCTY and in a number of cases it detects property violation without generating the entire state space, which makes the combined algorithm on-the-fly.

However, even though the algorithm is not asymptotically optimal, in practice it is hardly a problem when it comes to performance – the bottlenecks can be found elsewhere.

```
$ divine-mc owcty mutex_peterson.naive.dve
 initialize...             |S| = 81219
------------- iteration 0 -------------
 reachability...           |S| = 81219
 elimination & reset...    |S| = 59736
------------- iteration 1 -------------
 reachability...           |S| = 59736
 elimination & reset...    |S| = 59736
 =====================================
        Accepting cycle FOUND
 =====================================
 generating counterexample...      done
```

**Fig. 3.** Invocation of the tool for LTL model checking

## 3 Using the Tool

First and foremost, the model needs to be specified in the DVE modelling language and the property needs to be specified either as an LTL formula or as a Büchi automaton. We will present usage of the tool on a simple example of a mutual exclusion protocol. Source code of DVE specification of a single process of such a model can be found in Figure 1. The first LTL property we will use is $\mathbf{GF}c_0 \wedge \mathbf{GF}c_1$, which is a naïve formulation of the idea that the two processes are infinitely often in the critical section. An improved version of the formula that enforces fairness will be $(\mathbf{GF}(a_0 \wedge w_0) \rightarrow \mathbf{GF}c_0) \wedge (\mathbf{GF}(a_1 \wedge w_1) \rightarrow \mathbf{GF}c_1)$. The propositions $a$ and $w$ mean that the given process is active (when $a$ holds) and that it is waiting (when $w$ holds). First of these formulae is invalid (and the tool produces a counterexample), whereas the second one will be shown to hold for the model presented.

An example invocation of the tool for the model with 3 processes (and the formulae extended to 3 processes straightforwardly) can be seen in Figure 3. The counterexample could be browsed by running `divine-mc.simulator` on the produced `mutex_peterson.naive.trail`. The simulator is currently fairly rudimentary, but it still serves the purpose. When the same verifier command is used on the second formula, no counterexample is generated and the tool declares that an acceeping cycle has not been found, which means that the LTL property holds.

It can be seen that the input file to the verifier is a single DVE file that already contains a property process. Such a file could be written by hand (when the property has been specified as a Büchi automaton) or produced by `divine-mc.combine`, which takes a set of LTL formulae as input (in an `.ltl` file containing definitions of atomic propositions and the formulae – an example of such file containing the 2 discussed properties can be seen in Figure 2). The `divine-mc.combine` script will produce a single DVE file for each property, which can then be used as an input for the verifier.

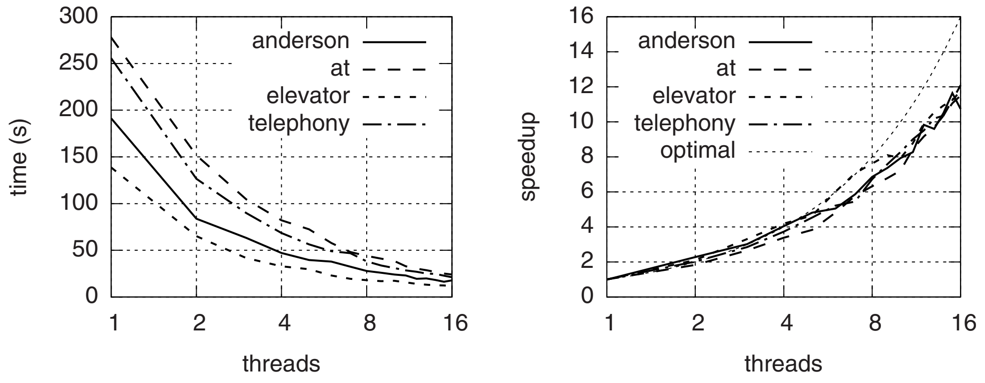


**Fig. 4.** Timing and speedup of reachability analysis

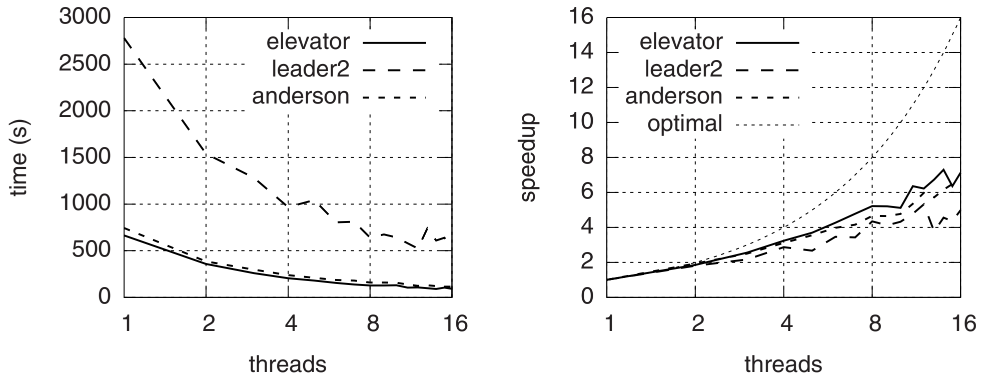


**Fig. 5.** Timing and speedup of LTL model checking, the algorithm used is OWCTY

## 4 Implementation

DiVinE Multi-Core is implemented on top of the POSIX Threads standard. Similarly to the distributed-memory approach, the state space is partitioned into parts, each thread being responsible for states of one of the parts. A thread maintains its own hashtable, explores successors of states of its part of the state space, and communicates with other threads by means of lock-free shared-memory message passing.

## 5 Experiments

The figures presented come from a 16-way AMD Opteron 885 (8 CPU units with 2 cores each) machine, with 64G of RAM, compiled with gcc 4.2.2 in 64-bit mode, using -O2. The models have been taken from a DVE model database [8]. Their descriptions and the verified properties can be found in the database: *anderson* is `anderson.6.dve` and `anderson.6.prop4.dve`, *elevator* is `elevator2.3.dve` and `elevator2.3.prop4.dve`, *at* is `at.5.dve`, *leader2* is `leader_election.5.prop2.dve` and finally *telephony* is `telephony.7.dve`. The property-less models have been used in reachability timing, whereas those containing LTL property (suffixed `propN.dve`) have been used for OWCTY timing.

More experimental data can be found on the tool webpage [6].

## 6 Future Work

To improve the usefulness of the tool we plan to implement a graphical interface for counterexample browsing, which would be much more intuitive than the current fairly rudimentary simulator available.

Moreover, we intend to further optimize the state space generator, which currently seems to be the main bottleneck of the tool – therefore the tool would benefit greatly from an improved interpreter.

Another future goal is to adapt and implement some of the known distributed memory partial order reduction techniques.

## References

1. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core ltl model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification (Tool Paper). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
3. Barnat, J., Ročkai, P.: Shared Hash Tables in Parallel Model Checking. In: Participant proceedings of the Sixth International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007), pp. 81–95. CTIT, University of Twente (2007)
4. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
5. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection (set based approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
6. DiVinE – Distributed Verification Environment, Masaryk University Brno, `http://anna.fi.muni.cz/divine-mc/`
7. Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the spin model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
8. Pelánek, R.: BEEM: BEnchmarks for Explicit Model checkers (February 2007), `http://anna.fi.muni.cz/models/index.html`
9. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. IEEE Symposium on Logic in Computer Science, pp. 322–331. IEEE Computer Society Press, Los Alamitos (1986)

# DiVinE-CUDA – A Tool for GPU Accelerated LTL Model Checking[*]

Jiří Barnat
Faculty of Informatics
Masaryk University
Brno, Czech Republic
barnat@fi.muni.cz

Luboš Brim
Faculty of Informatics
Masaryk University
Brno, Czech Republic
brim@fi.muni.cz

Milan Češka
Faculty of Informatics
Masaryk University
Brno, Czech Republic
xceska@fi.muni.cz

In this paper we present a tool that performs CUDA accelerated LTL Model Checking. The tool exploits parallel algorithm MAP adjusted to the NVIDIA CUDA architecture in order to efficiently detect the presence of accepting cycles in a directed graph. Accepting cycle detection is the core algorithmic procedure in automata-based LTL Model Checking. We demonstrate that the tool outperforms non-accelerated version of the algorithm and we discuss where the limits of the tool are and what we intend to do in the future to avoid them.

## 1 Introduction

Verification and validation became an important part of the design process. Unfortunately, the gap between the complexity of systems the current formal verification tools can handle and the complexity of systems built in practice is still quite wide. Therefore, any technique that accelerates the verification process is highly desirable. A possible way to reduce the delay due to the formal verification process is to accelerate the computation of verification tools using contemporary parallel hardware. Hardware platforms such as multi-core multi-cpu systems or many-core hardware accelerators, e.g. GPGPUs, have recently received a lot of attention in this aspect.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA [7]. Recently, it has been successfully used to accelerate formal verification process for selected settings. In [4] authors demonstrated significant speedup in the verification of probabilistic systems, while in [8, 9] CUDA has been used to accelerate disk-based model checking and state space generation. Let alone the CUDA technology, other many-core hardware acceleration platforms have been tried. For example, an implementation of FPGA accelerated Mur$\varphi$[13] verification tool has been reported in [10].

In this paper we introduce a new CUDA accelerated verification tool for model checking formulas of Linear Temporal Logic (LTL). The problem of LTL model checking is well established problem in the formal verification community. Computationally the problem reduces to the problem of detection of an accepting cycle in a directed graph [14]. The new tool builds upon the DiVinE [2] framework, hence the name of the tool is *DiVinE CUDA*.

## 2 DiVinE CUDA Algorithmics

DiVinE-CUDA employs algorithm MAP [5] for accepting cycle detection. The algorithm is, however, formulated as a repeated matrix-vector product procedure [3] in order to efficiently utilize CUDA archi-

[*]This work has been partially supported by Czech Science Foundation grants No. 201/09/P497, 102/09/H042 and by Academy of Sciences of Czech Republic grant No. 1ET408050503.

tecture. The idea of the MAP algorithm is as follows. Given a directed graph with accepting vertices, the algorithm impose ordering on accepting vertices and repeatedly computes the maximal (w.r.t. the ordering) accepting predecessor $map(u)$ for every accepting vertex $u$ in the graph. If the algorithm detects an accepting vertex that is its own maximal accepting predecessor, then the vertex lies on an accepting cycle and the algorithm terminates. In the other case, all accepting vertices that were maximal accepting predecessors for some other vertices are marked as non-accepting (because they do not lie on an accepting cycle) and the procedure is restarted (goes to the next iteration). The algorithm terminates either if accepting cycle is found or there are no more accepting vertices in the graph. From technical reasons we employ MAP algorithm on a transposed state space graph, note that graph transposition preserves the presence of accepting cycles.

The main computation demanding step of the algorithm is the computation of the maximal accepting predecessor for every accepting vertex. This is done by means of value propagation of accepting vertices along edges in the graph. If multiple values are propagated into a single vertex, the maximum among all the incoming values and the value of the vertex is computed and used for further propagation. Every vertex keeps the maximum value that has been propagated through the vertex. Once a fix-point is reached (no value can be improved), values of maximal accepting successors are computed.

In DiVinE CUDA tool it is the maximal accepting successor computation that is accelerated with CUDA device. In particular, relevant parts of the graph to be analyzed are represented in an adjacency matrix. Having the matrix, the value propagation can be realized as matrix-vector product [3] for computation of which the CUDA architecture is known to be extremely efficient [11].

When initiated the DiVinE CUDA tool proceeds as follows. It starts a thread that computes the adjacency matrix needed for CUDA processing. We use CSR (compressed sparse row) format to store the matrix. Note that we do not list all reachable states in the matrix, but only those that are in components containing some accepting vertices [12]. This feature significantly reduces the size of the matrix to be handled. (The size reduction is up to 20-30% of the full size in most cases). At the same time the tool runs a second thread that repeatedly performs CUDA accelerated accepting cycle detection on the part of the matrix that has been computed so far. If an accepting cycle is present in that part of the graph it is discovered before the full state space is generated. Therefore, DiVinE CUDA works on-the-fly.

## 3 Using the Tool

DiVinE-CUDA is a tool that stems from parallel and distributed LTL Model Checker DiVinE [2, 1]. As such, DiVinE-CUDA tool uses DiVinE native modeling language DVE [2]. In DVE modeling language the system to be verified is given as an asynchronous network of communicating finite automata. Transitions of every automaton in the network can be augmented with guards, buffered and unbuffered channel communication primitives, and variable updates.

The scheme of how the DiVinE CUDA tool should be used is given in Figure 1. Having prepared the model either directly as a *.dve* file or from a *.mdve* template using `divine.preprocessor` the user has to specify the property to be verified. The property can be given either directly as a property automaton (also known as never claim automaton) in the model file, or as (a set of) LTL formula(s) in a separate file, in which case the files have to be further processed by `divine.combine` tool to get a model file with the property automaton.

The next step in the verification process is to produce precompiled version of the model using `divine.precompile` tool. Precompiled version of the model (file with extension *.dveC*) is actually a dynamically linked library containing functions to generate states of the model with specification. Fi-

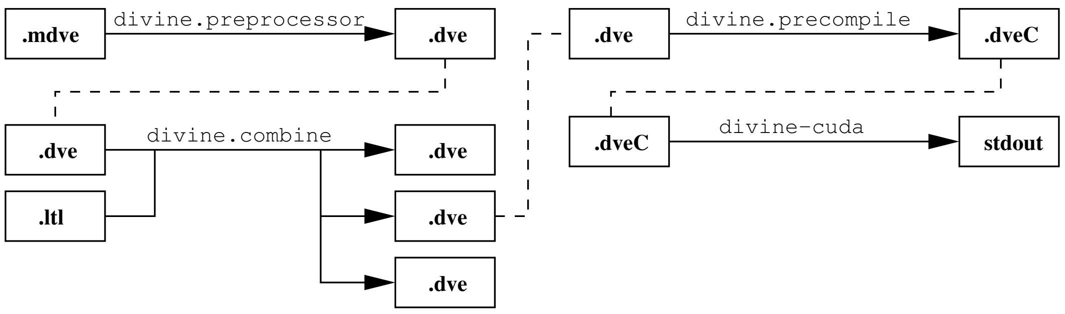


Figure 1: DiVinE CUDA work-flow.

nally, the precompiled representation of the model is used as an input for the `divine-cuda` tool itself.

During the computation the tool reports periodically the numbers of generated states and transitions, numbers of MAP iterations and CUDA device calls made so far to the standard output. At the end the tool outputs whether an accepting cycle has been found, in which case the given model does not satisfy the specification, or whether no accepting cycle has been discovered, i.e. the specification is satisfied.

## 4 Experiments

To briefly evaluate our tool we compared our implementation of CUDA accelerated MAP algorithm with the existing algorithms implemented in the DiVinE-Cluster version 0.8.2 model checker. For the comparison we used selected DiVinE native models including leader election protocol, elevator cabin system, Peterson's and Anderson's solutions to mutual exclusion problem and dining philosophers. We tested both the models with specification error (with an accepting cycle) as well as models without a specification error. All the experiments were run on a Linux workstation equipped with two AMD Phenom(tm) II X4 940 Processors @ 3MHz, 8 GB DDR2 @ 1066 MHz RAM and NVIDIA GeForce GTX 280 GPU with 1GB of GPU memory.

Table 1 provides details on run-times of individual algorithm parts. As for the CUDA MAP algorithm, the total run-time includes the initialization time (not reported in the table), CSR construction time (*CSR time*), and time spent on CUDA computation (*CUDA time*). Note that the first iteration of CPU MAP is actually slower than construction of the CSR representation. This is because the first iteration of the CPU MAP not only generates the state space, but also computes first stable values of *map*. Just for curiosity we also compare the performance of the new tool with DiVinE Cluster tool running OWCTY Algorithm [6]. Algorithms MAP and OWCTY were running on a single core.

Table 2 gives a comparison of overall run-times for both valid and invalid model checking instances. Though, the overall speedup is not that significant, it is still impressive. We can also see that the burden of data preparation is huge compared to the CUDA processing itself.

## 5 Availability and Future Work

At the moment the tool cannot handle models for which the corresponding *reduced* matrix of the graph does not fit the memory of a single CUDA device, it lacks the ability of counterexample generation, and cannot employ multiple threads to compute the CSR representation in parallel. We intend to address all

| Model | *accepting cycle* | CUDA MAP | | | CPU MAP | | | | CPU OWCTY | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *CSR time* | *CUDA time* | *total time* | *1st iter. time* | *other iter. time* | *total time* | *# iter.* | *reachability time* | *total time* |
| elevator 1 | N | 26 | 7 | 34 | 44 | 56 | 100 | 16 | 24 | 41 |
| leader | N | 87 | 1 | 90 | 97 | 600 | 697 | 17 | 90 | 297 |
| peterson 1 | N | 105 | 6 | 113 | 175 | 270 | 445 | 16 | 110 | 188 |
| anderson | N | 31 | 7 | 39 | 64 | 51 | 115 | 5 | 33 | 113 |
| elevator 2 | Y | 33 | 1 | 35 | 50 | – | 50 | 1 | 41 | 177 |
| phils | Y | 45 | 1 | 47 | 295 | 102 | 397 | 5 | 180 | 576 |
| peterson 2 | Y | 25 | 5 | 31 | 173 | – | 173 | 1 | 114 | 404 |
| bakery | Y | 24 | 1 | 26 | 240 | – | 240 | 1 | 219 | 907 |

Table 1: Comparison of run-times (in seconds) for CUDA accelerated MAP algorithm, non-accelerated MAP algorithm and OWCTY algorithm.

| Models | CUDA MAP | CPU MAP | | CPU OWCTY | |
|---|---|---|---|---|---|
| | *total time* | *total time* | *CUDA MAP speedup* | *total time* | *CUDA MAP speedup* |
| non-accepting | 276 | 1357 | 4.92 | 639 | 2.32 |
| accepting | 139 | 860 | 6.19 | 2064 | 14.87 |
| both | 415 | 2173 | 5.24 | 2730 | 6.51 |

Table 2: The overall run-times in seconds, and speedup of the whole model checking procedure.

| | CUDA MAP | | | CPU MAP | | CPU OWCTY | |
|---|---|---|---|---|---|---|---|
| | *CSR time* | *CUDA time* | *total time* | | | | |
| **1 core**: | 386 + 29 = | | 415 | Total time: | 2 173 | Total time: | 2 730 |
| | | | | Speedup: | **5.24** | Speedup: | **6.51** |
| **2 cores**: | 193 + 29 = | | 222 | Total time: | 1087 | Total time: | 1365 |
| | | | | Speedup: | **4.87** | Speedup: | **6.15** |
| **4 cores**: | 97 + 29 = | | 126 | Total time: | 544 | Total time: | 683 |
| | | | | Speedup: | **4.32** | Speedup: | **5.42** |
| **8 cores**: | 49 + 29 = | | 78 | Total time: | 272 | Total time: | 342 |
| | | | | Speedup: | **3.48** | Speedup: | **4.38** |

Table 3: A hypothetical speedup of DiVinE CUDA w.r.t. multicore parallel algorithms. We suppose optimal (linear) speed-up for both parallel algorithms MAP and OWCTY and for the CSR construction phase of the CUDA MAP algorithm.

these issues in the next version of the tool. As for the run-times, we expect significant improvement due to parallel preparation of the CSR graph representation. See Table 3. As for the limit on the size of the verification problem, we plan to introduce sort of clever swapping mechanism of the matrix stored in the GPU memory and to extend the memory available by employ multiple CUDA devices.

DiVinE CUDA tool is freely available from DiVinE web pages [1] where we provide both download and install instructions as well as simple tutorial on using the tool.

## References

[1] J. Barnat, L. Brim & P. Ročkai (2008): *DiVinE Multi-Core – A Parallel LTL Model-Checker*. In: *Automated Technology for Verification and Analysis*, *LNCS* 5311. Springer, pp. 234–239.

[2] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai & P. Šimeček (2006): *DiVinE – A Tool for Distributed Verification (Tool Paper)*. In: *Computer Aided Verification (CAV 2006)*, *LNCS* 4144. Springer, pp. 278–281.

[3] J. Barnat, L. Brim, M. Češka & T. Lamr (2009): *CUDA accelerated LTL Model Checking*. Technical Report FIMU-RS-2009-05, Faculty of Informatics, Masaryk University.

[4] D. Bosnacki, S. Edelkamp & D. Sulewski (2009): *Efficient Probabilistic Model Checking on General Purpose Graphics Processors*. In: *Model Checking Software (SPIN 2009)*, *LNCS* 5578. Springer, pp. 32–49.

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

[6] I. Černá & R. Pelánek (2003): *Distributed Explicit Fair Cycle Detection (Set Based Approach)*. In: *Model Checking Software (SPIN'03)*, *LNCS* 2648. Springer, pp. 49–73.

[7] *NVIDIA CUDA Compute Unified Device Architecture*. `http://www.nvidia.com/object/cuda_home.html`, August 2009.

[8] S. Edelkamp & D. Sulewski: *Technical Report on GPU based Model Checking*. `http://www.tzi.de/~edelkamp/GPU_Technical.pdf`, August 2008.

[9] Stefan Edelkamp & Damian Sulewski (2009): *Parallel State Space Search on the GPU*. In: *International Symposium on Combinatorial Search (SoCS 2009)*.

[10] M. E. Fuess, M. Leeser & T. Leonard (2008): *An FPGA Implementation of Explicit-State Model Checking*. In: *16th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)*. IEEE Computer Society, pp. 119–126.

[11] Michael Garland (2008): *Sparse Matrix Computations on Manycore GPU's*. In: *Proceedings of the 45th annual conference on Design automation (DAC'08)*. ACM, pp. 2–6.

[12] A. L. Lafuente (2002): *Simplified distributed LTL model checking by localizing cycles*. Technical Report 00176, Institut für Informatik, University Freiburg, Germany.

[13] U. Stern & D. L. Dill (1997): *Parallelizing the Mur$\varphi$ Verifier*. In: O. Grumberg, editor: *Proceedings of Computer Aided Verification (CAV '97)*, *LNCS* 1254. Springer-Verlag, pp. 256–267.

[14] M. Y. Vardi & P. Wolper (1986): *An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)*. In: *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*. IEEE Computer Society Press, Washington, DC, pp. 332–344.

[1] `http://divine.fi.muni.cz/page.php?page=divine-cuda`

2009 International Workshop on High Performance Computational Systems Biology

# DiVinE 2.0: High-Performance Model Checking

J. Barnat, L. Brim and P. Ročkai
Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

***Abstract*—We present a tool for parallel enumerative LTL model-checking and reachability analysis. The tool brings model checking to high-powered multi-core systems, as well as high-performance clusters. Boasting pluggable modelling language framework, it is possible to leverage the available parallel algorithms for multiple problem domains, by using suitable input language.**

## I. Introduction

In modern computer science, there are many applications for model checking. No doubt, a number of applications is placing great demands on model checkers – the models have become very large and vast resources are required to verify the desired properties. This is the case, for example, if we want to apply model checking algorithms to verification of specific properties of a complex biological model in order to verify consistency of predictions of the system behaviour with experimental data [1].

However, even though contemporary hardware is capable of providing these resources, it is difficult to employ them efficiently in model checking: a tool is needed that would target high-performance parallel computers and clusters.

Historically, there have been two branches of DiVinE. First of those two, DiVinE Cluster [2], targeted distributed-memory environments through use of Message Passing Interface [3]. However, in 2007, we have found that the existing DiVinE implementation is inadequate for contemporary shared-memory multiprocessors. Parts of the tool have been rewritten specifically for shared-memory systems and released as a separate tool, DiVinE Multi-Core [4], [5].

In the course of its development, it became clear that the distributed-memory branch of DiVinE would benefit from improvements that have been made in DiVinE Multi-Core. The current version, DiVinE 2.0 builds on both those branches. The shared-memory capabilities give the tool high performance on modern multi-core machines. Moreover, it can take advantage, through distributed memory, of multiple such systems at once, using the large aggregated memory to handle very complex models that do not fit the physical memory of a single computer.

Apart from the now-classical DVE modelling language, DiVinE 2.0 comes with a flexible model input system that allows for additional modelling languages to be used. The input model may be either interpreted by one of the built-in interpreters, or through an external interpreter that can be provided by 3rd parties, or the model may be compiled by an external compiler and loaded using a binary interface into DiVinE, providing further performance benefits.

Full source code of the tool can be obtained from [6]. As of this writing, the latest available version is first beta of 2.0.

## II. Model-Checking Algortihm

DiVinE is based on automata-theoretic approach to LTL model checking [7]. The DVE input language lets the user specify processes in terms of extended finite automata. These processes are then composed asynchronously to obtain the system to be verified. This system is in turn synchronously composed with a property process (negative claim automaton) obtained from the verified LTL formula through a Büchi automaton construction.

The resulting finite product automaton is then checked for presence of accepting cycles (fair cycles), indicating non-emptiness of its accepted language – which coincides with invalidity of the verified LTL property.

It is the fair cycle detection that happens to be the most time-consuming part of the model-checking process. It is also very memory-intensive, since the complete product automaton needs to be stored in RAM, and for moderately complex systems, the product automaton can have tens or hundreds of millions of vertices. With certain algorithms (including the one employed by DiVinE), the memory requirements can be smaller when the considered system contains an error: only the part of the system that the algorithm explored until it discovered the error needs to be stored. Unfortunately, this does not help with error-free models – and it is in the nature of model checking, that erroneous models often evolve into bug-free ones.

Since fair cycle detection is such a resource-intensive task, it is only logical to apply parallel algorithms. Unfortunately, it also happens to be a task that is hard to parallelise efficiently. Nevertheless, as DiVinE shows, it is definitely possible to outperform the strictly serial algorithms, even though the latter have better theoretical complexity. Moreover, with ongoing hardware development, the performance gap between best parallel and best serial algorithms will only widen.

## III. Using the Tool

The input model can be provided in a DVE format, or, as outlined above, custom interpreters or pre-compiled binaries can be used. In case of a DVE model, the property needs to be specified either as an LTL formula or as a Büchi automaton.

978-0-7695-3809-9/09 $26.00 © 2009 IEEE
DOI 10.1109/HiBi.2009.10

```
$ divine verify beem-peterson.4.prop4.dve
 initialise...              |S| = 1119479
------------ iteration 1 ------------
 reachability...            |S| = 1119479
 elimination & reset...     |S| = 0
 ===================================
       Accepting cycle NOT found
 ===================================
```

Fig. 1. Invocation of the tool for LTL model checking.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 101 | 64.9 | 50 | 39.5 |
| 2 | 84 | 42 | 35.7 | (48.4) |
| 3 | 49.4 | 29.1 | 34.9 | (35.7) |
| 4 | 39.5 | 21.6 | 22.3 | (27.6) |
| 6 | 38.7 | 16.5 | 16.8 | (21.2) |
| 8 | 20.8 | 14.5 | 13.4 | (19.1) |
| 10 | 21.4 | 13.4 | 12.5 | (18.8) |

TABLE I
TIMING OF REACHABILITY ON DIFFERENT NUMBER OF MACHINES (1-10) AND DIFFERENT NUMBER OF THREADS PER MACHINE (1-4). THE TIMES ARE IN SECONDS.

An example invocation of the tool for a model of Peterson's mutual exclusion protocol with four processes can be seen in Figure 1. The LTL property states that a process enters the critical section infinitely often.

As shown in the figure, the input file to the verifier is a single DVE file that already contains a property process. Such a file could be written by hand (when the property has been specified as a Büchi automaton) or produced by `divine-mc.combine`, which takes a set of LTL formulae as input (in an `.ltl` file containing definitions of atomic propositions and the formulae). The `divine-mc.combine` script will produce a single DVE file for each property, which can then be used as an input for the verifier.

For external models, it is the responsibility of the interpreter or compiler in question to provide the full (i.e. including property) product automaton with marked accepting states – it is currently not possible for DiVinE to verify arbitrary LTL properties for models provided externally, although a limited support for such a mode is planned for a future release.

## IV. Implementation and Experiments

DiVinE 2.0 is based on the POSIX Threads [8] standard for shared memory parallelism, and on Message Passing Interface [3] for distributed memory computations. The state space is partitioned into parts, each thread of each MPI node being responsible for vertices of its assigned part of the state space. Each thread maintains its own hashtable, exploring successors of states of its own part of the state space, and communicates with other threads when foreign states are discovered.

To test the tool's scalability, we have executed a small experiment using 10 identical machines, each with 4 Intel Xeon 5130 cores and 16G of memory, interconnected using off-the-shelf gigabit ethernet. We have used simple reachability on a model with very small states and very fast successor generation to stress the parallel algorithm. The results are shown in Table I. When using 3 worker and 1 MPI thread per node, the tool can use all 10 machines without any major speed regressions – to the contrary the time required is dropping with increasing number of machines employed (though using 12 or 14 nodes does not bring any further speedup). However, this places almost 160G of RAM at the tool's disposal, using stock hardware, while maintaining interesting speedup.

With all 10 machines in the test cluster (total of 40 cores), the overall speedup is over 8 and using 4 machines (and therefore 16 cores) of the cluster gives speedup of 4.6. For reference, using 16-cores in a single shared-memory machine, the speedup obtained on this model was around 6. Of course, the cluster's commodity ethernet interconnect cannot match the internal bus of the 16-core shared memory system, which is also clearly reflected in the test results.

## V. Future Work

Work is being done on graphical counterexample browser, which would greatly improve usability. Moreover, state space reductions and further optimisation is planned, to push the size of verifiable models even further.

Moreover, a new modelling language is being designed to replace the aging DVE format, with both an interpreter and a compiler (for high-performance model checking). The new language aims to improve modelling flexibility to facilitate modelling of wider array of system types.

## Acknowledgment

The work of J. Barnat has been supported by GAČR grant No. 201/09/P497. L. Brim has been supported by GAČR grant No. 201/09/1389 and FP6 project No. NEST-043235 (EC-MOAN). The work of P. Ročkai has been supported by the Academy of Sciences grant No. 1ET408050503 and by Red Hat, Inc.

## References

[1] J. Barnat, L. Brim, I. Černá, S. Dražan, and D. Šafránek, "Parallel model checking large-scale genetic regulatory networks with divine," *Electron. Notes Theor. Comput. Sci.*, vol. 194, no. 3, pp. 35–50, 2008.

[2] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "DiVinE – A Tool for Distributed Verification (Tool Paper)," in *Computer Aided Verification*, ser. LNCS, vol. 4144/2006. Springer Berlin / Heidelberg, 2006, pp. 278–281.

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.1," 2008. [Online]. Available: http://www.mpi-forum.org/docs/mpi21-report.pdf

[4] J. Barnat, L. Brim, and P. Ročkai, "Scalable Multi-core LTL Model-Checking," in *Model Checking Software*, ser. LNCS, vol. 4595. Springer, 2007, pp. 187–203.

[5] ——, "DiVinE Multi-Core – A Parallel LTL Model-Checker," in *Automated Technology for Verification and Analysis*, ser. LNCS, vol. 5311. Springer, 2008, pp. 234–239.

[6] "DiVinE – Distributed Verification Environment, Masaryk University Brno." [Online]. Available: http://divine.fi.muni.cz/

[7] M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. IEEE Symposium on Logic in Computer Science*. Computer Society Press, 1986, pp. 322–331.

[8] IEEE, *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1995.

Fourth International Conference on the Quantitative Evaluation of Systems

# ProbDiVinE: A Parallel Qualitative LTL Model Checker*

Jiří Barnat, Luboš Brim, Ivana Černá, Milan Češka, and Jana Tůmová
Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic

## Abstract

*We introduce a parallel model checker for checking Markov decision processes against linear time properties. The model checker extends the parallel model checker DiVinE and supports verification of qualitative properties.*

## 1. Introduction

Probabilistic systems like Markov chains and Markov decision processes provide a reasonable semantics for systems that exhibit uncertainty. Model checking of probabilistic systems branches into qualitative and quantitative approach. In the qualitative setting it is checked whether a property holds with probability 0 or 1; in a quantitative setting it is verified whether the probability for a certain property meets a given lower or upper bound.

For probabilistic systems the state explosion problem is more critical than in the non-probabilistic case. Several methods that have been developed for non-probabilistic systems to avoid the state explosion, were adapted to probabilistic systems. For branching time logics these are the symbolic approach implemented in the model checker PRISM [8, 7] and the MDP model checker RAPTURE [3] which uses an iterative abstraction refinement. For linear time logic the most prominent partial order approach has been recently adapted as well [1] and implemented in the verification tool LiQuor [4].

Over the past decade, many techniques using distributed and/or parallel processing have been proposed to combat the computational complexity of non-probabilistic verification, model checking in particular. However, not much has been done in applying these techniques to the verification and analysis of probabilistic systems. A notable exception is the work on parallelizing the symbolic model checker PRISM [10].

In this short tool paper we introduce a parallel model checker ProbDiVinE for qualitative model checking of finite state Markov decision processes (MDPs) against LTL properties. We use the automata-theoretic approach [9, 5, 6] where qualitative LTL model checking of MDPs is reduced to the question whether the product automaton for a given MDP with a Büchi acceptance condition contains an accepting end component (AEC).

## 2. How the tool works

The tool implements a parallel adaptation of the algorithm of de Alfaro (dA) [2]. It computes the set of states that contain all accepting end components. In particular, the algorithm maintains an *approximation set* of states that may belong to an AEC. The algorithm repeatedly refines the approximation set by locating and removing states that cannot belong to an AEC, we call this a *pruning step*. The core of the algorithm is the set of conditions determining the states to prune.

The algorithm by de Alfaro was the only one from the existing sequential approaches that allowed for a reasonable parallelization. The other algorithms, like the one of Courcoubetis and Yannakakis (CY) and CY with recursive elimination (CY+RE) [5] are based on repeated decomposition of the underlying graph of product MDP into strongly connected components and elimination of states violating their ergodic consistency or other techniques that are inherently sequential. For convenience ProbDiVinE also provides these serial algorithms and can use them on a single machine.

The input language PROBDVE of ProbDiVinE is a modification of DiVinE's native language DVE. PROBDVE models systems as a composition of processes, which can change their states via probabilistic (`=>`) or non-probabilistic (`->`) transitions and can synchronize using channels. An example of a PROBDVE source-code for randomized solution to the dining philosophers problem is given in Figure 1.

A non-probabilistic transition may have a *guard* (a condition which has to be satisfied for the transition to be enabled), an *effect* (assignment to a variable), and a *sync* expression (for synchronization with another transition via channel). A probabilistic transition just determines probability of resulting state, which is given by weights assigned

*This work has been supported in part by the Academy of Sciences grant No. 1ET408050503.

```
byte fork[3]={0,0,0};  byte hungry[3]={1,1,1};
process Philosopher_0 {
state thinks, eats, want_L, want_R, has_L, has_R;
init thinks;
trans
  thinks =>       { want_L:1, want_R:1   },
  want_L -> has_L { guard  fork[0] == 0;
                    effect fork[0] =  1; },
  has_L  -> eats  { guard  fork[1] == 0;
                    effect fork[1] =  1; },
  has_L  -> thinks{ guard  fork[1] == 1;
                    effect fork[0] =  0; },
  want_R -> has_R { guard  fork[1] == 0;
                    effect fork[1] =  1; },
  has_R  -> eats  { guard  fork[0] == 0;
                    effect fork[0] =  1; },
  has_R  -> thinks{ guard  fork[0] == 1;
                    effect fork[1] =  0; },
  eats   -> thinks{ effect hungry[0]= 0; };
}
process Philosopher_1 { ... }
...
system async;
```

**Figure 1. Example of** PROBDVE **source-code.**

to states. For example $s => \{s:3, t:2, u:2\}$ means that from the state $s$ system results in the state $s$ with probability $\frac{3}{7}$, in the state $t$ with the probability $\frac{2}{7}$, as well as in the state $u$.

ProbDiVinE is build on the top of the DiVinE library that offers common functions needed to develop a parallel or distributed enumerative model checker. The only extension to the library that was necessary, was the extension of the state generator to a probabilistic state generator, hence, it can handle probabilistic transitions of PROBDVE as introduced above. For the structure of ProbDiVinE implementation and connection to DiVinE see Figure 2.

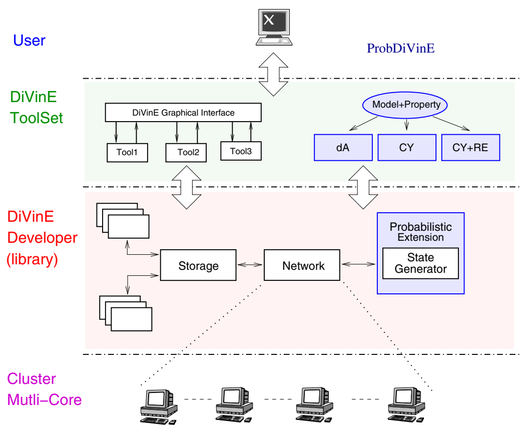


**Figure 2. ProbDiVinE structure.**

ProbDiVinE is currently operated in command-line mode. The input is the model given as a `.probdve` file and the LTL formula given as an `.ltl` file. Further parameters include the algorithm to be used, number of workstations involved in the computation etc. Details are given in the documentation. The tool can be downloaded together with the DiVinE tool from the page `http://anna.fi.muni.cz/divine/probdivine`.

Our initial experiments confirmed good scalability on many examples. In several cases the sequential algorithms were much faster than the parallel algorithm, however, once the internal memory has been exhausted only the parallel algorithm was able to finish the computation.

Our intention is to extend ProbDiVinE to quantitative LTL model checking by using a distributed linear solver, to add the possibility of checking reward properties, as well as to build a suitable graphical user interface.

## References

[1] C. Baier, M. Größer, and F. Ciesinski. Partial order reduction for probabilistic systems. In *Proc. QEST'04*, pages 230–239. IEEE, 2004.

[2] A. Bianco and L. de Alfaro. Model Checking Probabilistic and Nondeterministic Systems. In *Proc. FSTTCS'95*, LNCS, pages 499–513. Springer, 1995.

[3] B.Jeannet, P.dArgenio, and K.G. Lar. RAPTURE: A tool for verifying Markov Decision Processes. In *Proc. Tools Day/CONCUR02*, pages 84–98. 2002.

[4] F. Ciesinski and C. Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. Proc. QEST'06, 2006.

[5] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[6] L. de Alfaro. *Formal Verification of Stochastic Systems*. PhD thesis, Stanford University, Department of Computer Science, 1997.

[7] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. TACAS'06*, LNCS, pages 441–444. Springer, 2006.

[8] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *STTT*, 6(2):128–142, 2004.

[9] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. FOCS 1985*, pages 327–338. IEEE, 1985.

[10] Y. Zhang, D. Parker, and M. Kwiatkowska. Grid-enabled probabilistic model checking with PRISM. In *Proc. AHM'05*, 2005.

Quantitative Evaluation of SysTems

# ProbDiVinE-MC: Multi-Core LTL Model Checker for Probabilistic Systems*

Jiří Barnat, Luboš Brim, Ivana Černá, Milan Češka, and Jana Tůmová
Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic

## Abstract

*We present a new version of* PROBDIVINE *– a parallel tool for verification of probabilistic systems against properties formulated in linear temporal logic. Unlike the previous release [1], the new version of the tool allows for both quantitative and qualitative model-checking. It is also strictly multi-threaded, therefore, protects users from unwanted burden of parallel computing in a distributed-memory environment.*

## 1. Introduction

Model-checking of probabilistic systems splits into qualitative and quantitative branch. While in qualitative verification the procedure decides whether the property holds with probability one or less, in the quantitative approach the procedure decides whether the probability of a certain property meets a given lower or upper bound.

There are several model-checking tools available for qualitative and quantitative verification of probabilistic systems. Similarly to the non-probabilistic case, the tools suffer from the well known state space explosion problem. Therefore, they apply various techniques to fight it and to extend the applicability of the tool. For branching time logic, we should mention PRISM [6] – a model-checker that uses symbolic state space representation, or MDP model-checker RAPTURE [2] that builds upon an automatic abstraction refinement and essential state reduction techniques. For linear time logic, the standard partial order reduction was implemented in LiQuor [3]. An alternative approach for coping with the state space explosion is also to investigate alternate computer architectures such as parallel or distributed systems. As an example of this approach we refer to the previous release of PROBDIVINE [1] that was capable of utilizing aggregate memory of computers in a network of interconnected workstations.

In this paper we consider automata-theoretic approach for enumerative LTL model-checking of probabilistic systems represented as Markov decision processes (MDP). The property to be verified is negated and the negation is expressed as a semi-deterministic Büchi automaton [4] which is then multiplied with an MDP of the system under consideration into a product Büchi MDP. Having the graph of the Büchi MDP, the problem of *qualitative* verification is reduced to the problem of detection of a reachable accepting ergodic component (AEC) in the graph [5]. The detection of AECs may be done in time almost linear with respect to the size of the product MDP. Therefore, the main factor that limits the applicability of a qualitative tool are the memory requirements for storing visited states of the MDP. However, this is not the case in the *quantitative* approach where the detection of AECs is further succeeded by a transformation of the product MDP into a set of linear inequalities (linear program) to be solved by some linear programming solver. Experience shows, that the solver is the bottleneck point as finding the optimal solution to the linear program is rather expensive in time as compared to the AEC detection. What our tool does to solve the quantitative problem efficiently is that it applies several subtle techniques to decompose the linear program into many smaller ones and uses parallel calls to the solver to partially remedy the limiting computational time factor.

## 2. ProbDiVinE-MC

First, we would like to state explicitly what are the differences between the previous release, referred to as PROBDIVINE, and the new version, reffered to as PROBDIVINE-MC. PROBDIVINE [1] allows users to perform parallel verification of qualitative aspects of probabilistic systems using distributed-memory environment, i.e. using aggregate power of computers in a cluster. On the other hand, PROBDIVINE-MC allows users to perform both *parallel qualitative* and *parallel quantitative* verification of probabilistic systems using shared-memory environment. Shared-memory parallelism became popular in recent years mainly due to the general availability of multi-cored CPUs and due to the fact that unlike the distributed-memory applications, shared-memory applications are eas-

*This work has been supported in part by the Czech Science Foundation grant No. GA201/06/1338 and by the Academy of Sciences of the Czech Republic grant No. 1ET408050503.

978-0-7695-3360-5/08 $25.00 © 2008 IEEE
DOI 10.1109/QEST.2008.29

| | # inequalities for LP solver | | | % of the whole graph | |
|---|---|---|---|---|---|
| Model | whole graph | reduced graph | largest subproblem | reduced graph | largest problem |
| Cons | 55734 | 52776 | 30774 | 94.7 | 55.2 |
| Phils | 2817561 | 184604 | 246 | 6.6 | Almost 0 |
| Stabi | 6897480 | 5983080 | 0 | 86.7 | 0 |
| Leads | 8800096 | 5678656 | 0 | 64.5 | 0 |
| Crypts | 8954217 | 132183 | 0 | 1.5 | 0 |

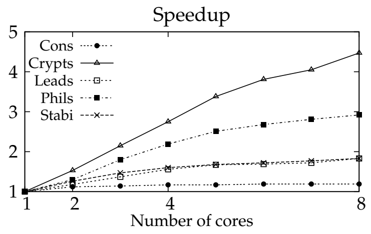


**Figure 1. a) Size of LP problem with respect to used reduction techniques b) Overall speedup.**

ier to be used by inexperienced users. Also the computational requirements of the quantitative analysis simply render shared-memory parallelism more convenient. Both ProbDiVinE and ProbDiVinE-MC use the ProbDVE input language [1].

For the construction of the tool, we have analyzed and reviewed the process of quantitative model checking phase by phase and pointed out several algorithmic modifications that have significant impact on a general performance of a quantitative verification tool. We also identified independent subtasks within individual phases of the process and apply parallel processing when possible. In particular, the tool employs our parallel technique to identify and remove states of MDP that are irrelevant with respect to the solution of the verification process. Then it applies a parallel algorithm for the detection of strongly connected components to partition the global LP problem into independent subproblems, solves the subproblems in parallel, and computes the global solution from the solutions to the subproblems. We have also identified a particular type of a subproblem for which the solution can be derived directly from the inequalities without using a call to an external LP solver.

## 3. Experimental evaluation

We have implemented the tool using the DiVinE Library and generally available LP solver `lpsolve`. We run a set of experiments on machines equipped with Intel Xeon 5130 and AMD Opteron(tm) 885 processors allowing us efficiently measure the performance of the tool when using 1 to 8 threads. In this paper we report on two different experiments related to quantitative verification only.

The table in Figure 1.a) captures the size of the linear programming problem (the number of inequalities to be solved by an LP solver) for five different models before and after the application of our reduction techniques. The first column (whole graph) gives the size before any reduction, the second column (reduced graph) gives the size when redundant inequalities were removed, and the third column (largest subproblem) gives the maximal size of a subproblem solved by an LP solver. Note that in some cases (largest subproblem = 0) we were able to find the global solution without calling LP solver at all. The size of the problem plays a crucial role in the performance of the tool. For example, the time consumed by the `lpsolve` to solve the model of Philosophers (Phils) was more than 2 days in the case of the whole graph, 45 minutes in the case of the reduced graph, and only 38 seconds, when all our reduction techniques were applied.

The verification process involves the parallel detection of AECs in an implicitly given graph, parallel detection and removal of redundant inequalities in the linear program, parallel decomposition of the linear program into subprograms, and all the concurrent calls to the LP solver. Figure 1.b) reports on the overall speed-up in the verification process we achieved using our tool on various number of CPU cores. We claim that our approach is quite successful as overall runtimes tend to decrease as more CPU cores are used. The tool is available at `http://anna.fi.muni.cz/probdivine`.

## References

[1] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Proc. of QEST'07*, pages 215–216. IEEE Computer Society, 2007.

[2] B.Jeannet, P. de Argenio, and K. Larsen. RAPTURE: A tool for verifying Markov Decision Processes. In *Proc. Tools Day / CONCUR'02. Tech.Rep. FIMU-RS-2002-05*, pages 84–98. MU Brno, 2002.

[3] F. Ciesinski and C. Baier. LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems. In *Proc. of QEST'06*, pages 131–132. IEEE Computer Society, 2006.

[4] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[5] L. de Alfaro. *Formal Verification of Stochastic Systems*. PhD thesis, Stanford University, Department of Computer Science, 1997.

[6] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of TACAS'06*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.