Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Multi-core and/or Symbolic Model Checking

Tom van Dijk , Alfons Laarman, Jaco van de Pol

7 pages

ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Multi-core and/or Symbolic Model Checking

Tom van Dijk[1 *], **Alfons Laarman**[1], **Jaco van de Pol**[1 †]

[1]Formal Methods and Tools, Department of EEMCS, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract: We review our progress in high-performance model checking. Our *multi-core model checker* is based on a scalable hash-table design and parallel random-walk traversal. Our *symbolic model checker* is based on Multiway Decision Diagrams and the saturation strategy. The LTSmin tool is based on the PINS architecture, decoupling model checking algorithms from the input specification language. Consequently, users can stay in their own specification language and postpone the choice between parallel or symbolic model checking. We support widely different specification languages including those of SPIN (Promela), mCRL2 and UPPAAL (timed automata). So far, multi-core and symbolic algorithms had very little in common, forcing the user in the end to make a wise trade-off between memory or speed. Recently, however, we designed a novel multi-core BDD package called Sylvan. This forms an excellent basis for scalable parallel symbolic model checking.

Keywords: multi-core model checking, symbolic model checking, scalability, hash-table, Binary Decision Diagrams, parallel algorithms

1 Introduction

In order to cope with the exuberant time and memory consumption of model checking, we have extensively experimented with two so far incompatible strategies. First, sets of state vectors can be represented very concisely as Multiway Decision Diagrams, leading to symbolic model checking of systems with astronomic state spaces. Second, explicit state model checkers get orders of magnitude faster by exploiting contemporary multi-core hardware, given scalable data structures and parallel algorithms. Traditionally, the two approaches are largely incompatible. Based on the problem at hand, users had to choose among symbolic and explicit model checkers, each with their own specification language.

In this invited contribution, we review our progress in high-performance model checking. We discuss a scalable hash-table design, enhanced with various scalable state compression techniques. Used by parallel random traversal, this design forms the key factor of a scalable model checker that can handle 10^{10} states in 10 minutes on a 48-core machine with 128 GB memory.

Simultaneously, we re-engineered several model checking algorithms and enhancements, such as LTL model checking with partial-order reduction, and symbolic model checking of μ-calculus properties. Our tool LTSmin offers all these features as language-independent building blocks, via the on-the-fly interface PINS. Besides a *partitioned* next-state function, this provides a minimal amount of structural model information.

[*] Supported by the NWO project MaDriD, grant nr. 612.001.101

[†] Corresponding author, vdpol@cs.utwente.nl

We implemented PINS language modules for Promela, mCRL2 and UPPAAL, so users can benefit from multi-core or symbolic verification of models in their preferred specification language. Still, the multi-core and symbolic algorithms are incompatible. In the end, one must choose between parallel or symbolic algorithms for each application.

Recently, we implemented a multi-core scalable BDD package, Sylvan. Its main data structures extend our scalable hash-table design. It has a parallel implementation of the BDD operation for relational product. As a consequence, we provide an excellent starting point for a truly scalable multi-core symbolic model checker; we already observed a speedup of 32 on 48 cores.

2 Exposing Transition Locality for State Space Exploration

Reachability analysis is the core algorithm shared by all model checkers. Given a set of states S, an initial state s_0, a transition relation R and a goal/error state t, the question is whether t is reachable, i.e. whether $s_0 R^* t$. Usually R is given only implicitly, by a next-state function that computes the successors of a given state.

We mainly consider the verification of asynchronous concurrent systems, consisting of a number of components, whose transitions are interleaved non-deterministically. In this case, a state consists of a vector of N values, representing the variables of all components, so $S = S_1 \times \cdots \times S_N$. The transition relation can be partitioned disjunctively, so $R = \bigcup_{i=1}^{M} R_i$. To cope with the memory and time demands of model checking, many algorithms exploit the *transition locality* in asynchronous systems: Typically R_i depends only on a small part of the state vector, by reading or modifying the corresponding variables. This is encoded in an $M \times N$ dependency matrix $D_{M\times N}$, where $D[i, j]$ represents that transition group i depends on state variable j.

For full flexibility, we want to decouple high-performance model checking algorithms from particular specification languages. In order to exploit transition locality, the interface should expose some model structure. Therefore, we proposed the PINS interface as the basic architecture for our LTSmin tool [BPW10, LPW11a], Figure 1. Every language module should provide:

- The length of the state vector $S_1 \times \cdots \times S_N$,
- A couple of next-state functions $R_i : S \to \mathscr{P}(S)$,
- The dependency matrix $D_{M\times N}$ (a syntactic overapproximation is sufficient).

LTSmin provides language modules to support process algebra with data (mCRL2 [GKS+11]), state based languages (Promela [Hol97, BL12] and DVE, used for benchmarking), and UPPAAL's timed automata [BDL+11, DLL+12]. It provides analysis tools based on distributed reachability (run on a cluster of workstations with a network connection), multi-core reachability (multiple processors with shared memory), and symbolic reachability tools (based on Multiway Decision Diagrams). It also has backend tools for model checking LTL and μ-calculus properties, or generating the complete state space, minimizing it modulo some equivalence, and storing it in a compressed file.

Due to this flexible setup, several optimization layers are provided as PINS2PINS wrappers, serving all combinations of languages and analyses. We just mention transition caching, state variable reordering, transition regrouping, and partial-order reduction [GW94, Pat11].

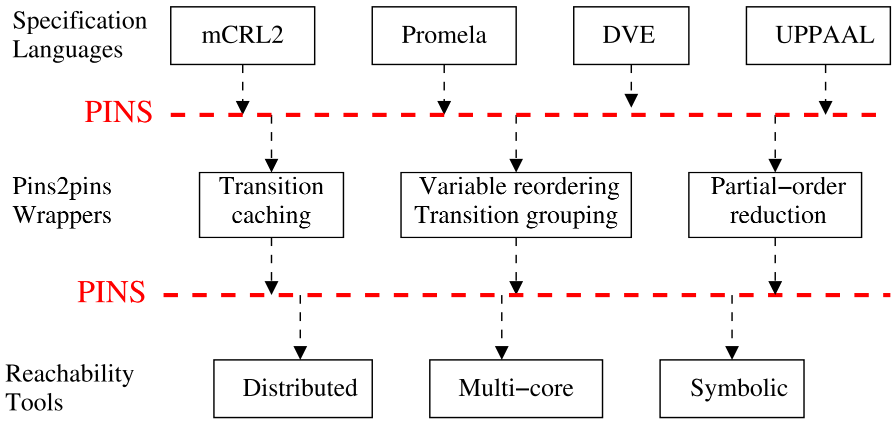

Figure 1: The PINS architecture of the LTSmin toolset

3 Multi-core Reachability and Liveness: Speedup and Compression

The essential data structure in explicit reachability algorithms is a hash-table containing the set of visited states. Before exploring newly generated states, they must be checked against this set. This avoids redundant re-exploration, or even non-termination in case of graphs with cycles.

We have adapted basic reachability analysis to multi-core computers, consisting of a number of processors and a large shared memory. To lower the latency penalty for accessing main memory, processors are usually equipped with hardware caches. Our basic parallel reachability scheme is to store the visited set in a single shared hash-table. Each worker maintains its private set of states to be explored. As a standard load balancing mechanism, when some worker runs out of work it steals some states from the set of another worker. So a scalable multi-core hash-table implementation is essential for parallel reachability. This is non-trivial, because hash-tables generate random memory accesses, leading to a lot of hardware cache misses. Also, to prevent concurrent reading and writing to the same address, some form of protection is asked for.

Based on the observation that for reachability analysis the set of visited states grows monotonically, we carefully crafted [LPW10] a lockless scalable hash-table with only one operation: `find_or_put(s):bool`. It guarantees that s will be in the table, and indicates whether it is new. Its efficiency is based on a number of design decisions:

- Store the hash values in a bucket array, separate from the state vectors. This avoids comparing long state vectors, except when their hash values collide.
- On hash collisions, use the next or previous locations within the hash bucket array (open addressing with bilinear probing), in order to stay on the same hardware cache line as long as possible.
- Use the primitive `compare_and_swap` to modify hash-buckets atomically, and reserve one bit to indicate that the accompanying state vector slot is being written.

Based on this scalable hash-table design, multi-core reachability analysis runs with near-optimal linear speedup for a wide range of benchmark models in various specification languages.

This can be extended to liveness analysis, by using parallel Nested Depth-First Search. Basically, each worker runs an independent NDFS procedure to detect accepting cycles in a Büchi automaton, coding for counter examples to an LTL property. Parallel search provides a considerable (even superlinear) speedup when accepting cycles exist (bug hunting). Intricate schemes for sharing information between workers deliver speedups also in the absence of accepting cycles [LLP+11, EPY11, ELPP12]. We observed a maximum speedup of 43 on 48 cores, and an average speedup of 17 among all 63 benchmarks of >1M states from the BEEM database [Pel07].

Having addressed time requirements, we next focused on the memory consumption of the set of visited states. Due to transition locality in asynchronous models, successive state vectors have a lot of similarity. This allows for impressive data compression. The key idea is to split state vectors in equal halves: $v = v_1 v_2$. Instead of storing all vectors v explicitly, we store them as pairs of integers (k_1, k_2), where k_j is an index in the table of values of v_j. Note that if v_1 and v_2 assume K different values, then the universe of v is bounded by K^2. Conversely, in the optimal case, a state space of N vectors of length L can be stored as $2N$ integers (pairs of indices) and 2 tables of $\sqrt{N}$ values of length $\frac{1}{2}L$. This trick can be repeated recursively to store the table of subvectors v_j, etc. We call the resulting technique *tree compression* [BLPW09, LPW11b].

We often observe a memory consumption close to the optimum. With incremental tree insertion, the required additional computations do not even slow down the multi-core performance. The reason is that compression also reduces the traffic over the main memory bus, increasing the *arithmetic intensity* of our application. So state compression comes for free indeed [LPW11b].

We implemented additional techniques to further reduce the memory footprint of the compressed tree, in particular a multi-core implementation [VL11] of Cleary Tables [Cle84]. Here, part of the hashed key is not even stored at all, but can be deduced from the actual location of the hash bucket at the expense of a small number of administration bits per bucket.

4 Symbolic Model Checking for Explicit State Languages

Even more state compression can be achieved with decision diagrams [Bry92]. In Multiway Decision Diagrams, a state vector is stored as a path in a Directed Acyclic Graph (DAG). A node in the diagram represents a set of vectors; an outgoing edge labeled v_i to t represents the set of vectors that start with v_i and have a suffix in t. Conciseness follows from the fact that different paths can share nodes: the number of paths can be exponentially larger than the number of nodes.

Interestingly, set-operations on MDDs (intersection, union) can be performed in polynomial time. As a consequence, large sets of states can be manipulated very efficiently. In particular, the relational product of a symbolic relation $R(v, v')$ applied to a set of state vectors $S(v)$ can be computed in polynomial time in the MDD representation of R and S. Symbolic model checking [BCM+90] is implemented by repeated application of the relational product.

The efficiency of symbolic model checking depends on the order of variables in the state vector, and on the exploration strategy. We implemented a static variable reordering heuristic, and several existing strategies to compute the fixed point of $R_1 \cup \cdots \cup R_M$:

- Breadth-First: compute $(R_1 \cup R_2 \cup \cdots \cup R_M)^*$ in a straightforward way;
- Chaining: compute $(R_1; R_2; \cdots; R_M)^*$, where $R_1; R_2$ denotes relation composition $R_2^{=} \circ R_1^{=}$;
- Saturation: compute $((((R_1^*); R_2)^*; R_3)^*; \cdots; R_M)^*$.

As shown by Ciardo [CLM07], saturation tends to minimize the peak memory of symbolic reachability, keeping the intermediate MDDs small. LTSmin implements the saturation strategy for the PINS interface [BP08, Sia12]. The individual relations R_i are learned on-the-fly. This provides symbolic model checking for explicit state languages. For instance, we can now symbolically explore a Promela model of 60 Dining Philosophers with 7.3×10^{41} states in 0.3 seconds (actually, compilation and static variable reordering takes much longer).

5 Sylvan: Multi-core BDDs for Parallel Symbolic Model Checking

Recently, we built a multi-core BDD package, called Sylvan [vD12, DLP12]. We will discuss the parallelization strategy in Sylvan, after first discussing its multi-core data structures.

The basic data structures in a BDD package are the Unique Table and the Computed Table. The Unique Table is a hash-table storing each BDD node once, to ensure that the DAG remains maximally shared. As a consequence, checking equality of two BDDs reduces to a simple pointer comparison. The Computed Table is needed to implement the BDD operators in polynomial time. Computing an operation O on BDDs B and C typically proceeds as follows:

$$O(\mathit{if}(p,B_1,B_2),\mathit{if}(p,C_1,C_2)) = \dots \mathit{if}(p,O(B_1,C_1),O(B_2,C_2))\dots$$

In order to ensure that the intermediate results $R_i := O(B_i,C_i)$ are computed only once, triplets (B_i,C_i,R_i) are cached in the Computed Table when they are computed the first time, so they can simply be looked up when needed a second time.

Sylvan stores BDD nodes in a single shared Unique Table, extending the lockless hash-table design explained before. Since BDD nodes come and go, we implemented parallel garbage collection based on reference counting. Really deleting elements would invalidate linear probing; they are replaced by tombstones instead, to be reused by new insertions. All workers assist during a parallel garbage collection phase. The parallel Computed Table is also globally shared. Here, in case of collisions, new results simply overwrite older ones.

The parallelization strategy of Sylvan is fairly simple: to apply some operation on BDDs, each worker traverses the BDDs independently. Intermediate results are stored in a *shared* Computed Table, avoiding recomputation of the same result by multiple workers. We rely on random traversal to ensure that workers traverse different parts of the BDDs, thus avoiding that all workers start computing the same piece of work.

An alternative implementation relies on Wool [Fax08], a work-stealing load balancer for fine-grained task parallelism. In Sylvan, a single task checks the Computed Table, spawns some recursive subtasks, creates the result node in the Unique Table, and stores a new triplet in the Computed Table. Using Wool gave a slight performance benefit over pure random traversal.

Our initial experiments showed a relative speedup of BFS-based symbolic reachability up to 32 on a 48-core machine (a speedup of 10 compared to single-threaded BuDDy). Some other models show only a very limited speedup. There is still plenty of work to do: improve parallel garbage collection, parallelize Multiway Decision Diagrams and experiment with saturation, as it is the best known sequential exploration strategy in many cases.

We view Sylvan as an excellent basis for further research into a scalable BDD package, with applications to symbolic model checking and beyond. This is a necessary condition to transfer the successful BDD technology forward to contemporary multi-core hardware.

Acknowledgements: Stefan Blom initiated the distributed and symbolic backends of LTSmin. Elwin Pater implemented partial-order reduction, variable reordering, LTL and mu-calculus model checking. Jeroen Ketema and Tien-Loong Siaw improved the symbolic backend by implementing saturation and guided exploration. Michael Weber contributed greatly to the LTSmin architecture, and the conception of the multi-core hash-table.

Bibliography

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Symp. on Logic in Computer Science (LICS 1990)*. Pp. 428–439. IEEE Computer Society, 1990.

[BDL$^+$11] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, W. Yi. Developing UPPAAL over 15 years. *Software Practice and Experience* 41(2):133–142, 2011.

[BL12] F. I. van der Berg, A. W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In Heljanko et al. (eds.), *Proc. PASM/PDMC 2012*. Electronic Notes in TCS (accepted). 2012.

[BLPW09] S. C. C. Blom, B. Lisser, J. C. van de Pol, M. Weber. A Database Approach to Distributed State-Space Generation. *J. of Logic and Computation* 21(1):45–62, 2009.

[BP08] S. C. C. Blom, J. C. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In Fitzgerald et al. (eds.), *Proc. Theoretical Aspects of Computing (ICTAC 2008)*. LNCS 5160, pp. 81–95. Springer, 2008.

[BPW10] S. Blom, J. C. van de Pol, M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Computer Aided Verification (CAV 2010)*. LNCS 6174, pp. 354–359. 2010.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys* 24(3):293–318, 1992.

[Cle84] J. G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Trans. Computers* 33(9):828–834, 1984.

[CLM07] G. Ciardo, G. Lüttgen, A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design* 31(1):63–100, 2007.

[vD12] T. van Dijk. Parallel Implementation of BDD operations for Model Checking. Master's thesis, University of Twente, April 2012.

[DLL$^+$12] A. E. Dalsgaard, A. W. Laarman, K. G. Larsen, M. C. Olesen, J. C. van de Pol. Multi-Core Reachability for Timed Automata. In Jurdziński and Ničković (eds.), *Formal Modeling and Analysis of Timed Systems (FORMATS)*. LNCS 7595, pp. 91–106. 2012.

[DLP12] T. van Dijk, A. W. Laarman, J. C. van de Pol. Multi-Core BDD Operations for Symbolic Reachability. In Heljanko et al. (eds.), *Proc. PASM/PDMC 2012*. Electronic Notes in TCS (accepted). 2012.

[ELPP12] S. Evangelista, A. W. Laarman, L. Petrucci, J. C. van de Pol. Improved Multi-Core Nested Depth-First Search. In Chakraborty and Mukund (eds.), *Proc. Automated Technology for Verification and Analysis (ATVA 2012)*. LNCS 7561, pp. 269–283. Springer, 2012.

[EPY11] S. Evangelista, L. Petrucci, S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In Bultan and Hsiung (eds.), *Proc. Automated Technology for Verification and Analysis (ATVA 2011)*. LNCS 6996. Springer, 2011.

[Fax08] K.-F. Faxén. Wool – a work stealing library. *SIGARCH Computer Architecture News* 36(5):93–100, 2008.

[GKS$^+$11] J. F. Groote, J. Keiren, F. P. M. Stappers, W. Wesselink, T. A. C. Willemse. Experiences in developing the mCRL2 toolset. *Software Practice and Experience* 41(2):143–153, 2011.

[GW94] P. Godefroid, P. Wolper. A Partial Approach to Model Checking. *Information and Computation* 110(2):305–326, 1994.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23(5):279–295, 1997.

[LLP$^+$11] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, A. Wijs. Multi-Core Nested Depth-First Search. In Bultan and Hsiung (eds.), *Proc. Automated Technology for Verification and Analysis (ATVA 2011)*. LNCS 6996. Springer, 2011.

[LPW10] A. W. Laarman, J. C. van de Pol, M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In Sharygina and Bloem (eds.), *Proc. of the 10th IC on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2010.

[LPW11a] A. W. Laarman, J. C. van de Pol, M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In Bobaru et al. (eds.), *NASA Formal Methods*. LNCS 6617, pp. 506–511. Springer, Berlin, 2011.

[LPW11b] A. W. Laarman, J. C. van de Pol, M. Weber. Parallel Recursive State Compression for Free. In Groce and Musuvathi (eds.), *SPIN 2011*. LNCS 6823, pp. 38–56. Springer, 2011.

[Pat11] E. Pater. Partial Order Reduction for PINS. Master's thesis, Univ. of Twente, 2011.

[Pel07] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'07*. LNCS 4595, pp. 263–267. Springer, 2007.

[Sia12] T.-L. Siaw. Saturation for LTSmin. Master's thesis, University of Twente, 2012.

[VL11] S. van der Vegt, A. W. Laarman. A Parallel Compact Hash Table. In Vojnar (ed.), *Proc. of Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2011*. LNCS 7119, pp. 191–204. Springer, 2011.