LTSMIN: Distributed and Symbolic Reachability

Stefan Blom*, Jaco van de Pol, and Michael Weber

Formal Methods and Tools, University of Twente, The Netherlands
{sccblom,vdpol,michaelw}@cs.utwente.nl

In model checking, analysis algorithms are applied to large graphs (state spaces), which model the behavior of (computer) systems. These models are typically generated from specifications in high-level languages. The LTSMIN toolset[1] provides means to generate state spaces from high-level specifications, to check safety properties on-the-fly, to store the resulting labelled transition systems (LTSs) in compressed format, and to minimize them with respect to (branching) bisimulation.

1 Motivation: A Modular, High-Performance Model Checker

The LTSMIN toolset provides a new level of modular design to high-performance model checkers. Its distinguishing feature is the wide spectrum of supported specification languages and model checking paradigms. On the language side (Sec. 3.1), it supports process algebras (MCRL), state based languages (PROMELA, DVE) and even discrete abstractions of ODE models (MAPLE, GNA). On the algorithmic side (Sec. 3.2), it supports two main streams in high-performance model checking: reachability analysis based on BDDs (symbolic) and on a cluster of workstations (distributed, enumerative). LTSMIN also incorporates a distributed implementation of state space minimization, preserving strong or branching bisimulation.

For end users, this implies that they can exploit other, scalable, verification algorithms than supported by their native tools, without changing specification language. Our experiments (Sec. 4) show that the LTSMIN toolset can match, and often outperform, existing tools tailored to their own specification language.

From an algorithm engineering point of view, LTSMIN fosters the availability of benchmark suites across multiple specification languages and verification communities. This makes benchmarking studies more robust, by separating out language-specific issues, which is of separate scientific interest. The LTSMIN toolset integrates very well with existing third-party tools (Sec. 3.3), for the benefit of their users, and also for the independent certification of model checking results.

The technical enabler of the LTSMIN toolset is its PINS interface (Sec. 2). This general abstraction of specification languages places very few constraints on their features, evident by the variety of supported languages (Sec. 3.1) and algorithms. PINS still enables the algorithms to exploit the parallel structure inherent in many specifications. Several optimizations are implemented as generic PINS2PINS wrappers, abstracting

* This research has been partially funded by the EC project EC-MOAN (FP6-NEST 043235).

[1] http://fmt.cs.utwente.nl/tools/ltsmin/, current version: 1.5, available as open-source software.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 354–359, 2010.
© Springer-Verlag Berlin Heidelberg 2010

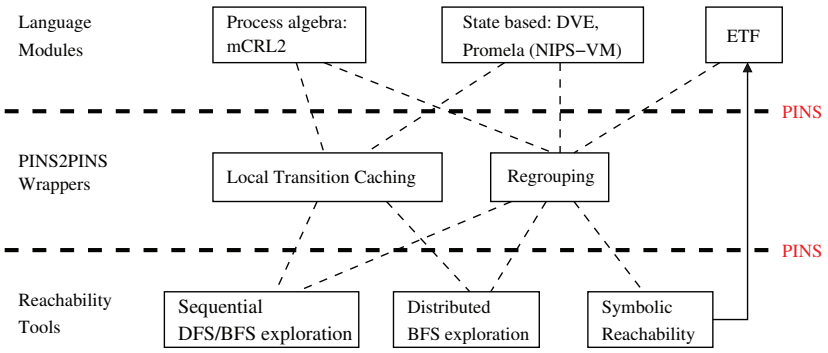

Fig. 1. Architectural Overview of PINS-Based Tools

from *both*, input language and the actual model checking paradigm. Thus, this opens new opportunities for research of reusable and composable implementations of model checking algorithms and optimizations.

2 Architecture: A Partitioned Next State Interface (PINS)

In order to separate specification languages from model checking algorithms, many enumerative, on-the-fly model checkers are based on some *next-state* interface. It provides transitions between otherwise opaque and monolithic states. For example, the OPEN/CÆSAR interface [1] has been underlying the success of the CADP toolkit [2].

The unifying concept in LTSMIN is an improvement of this interface, which we call PINS, an **I**nterface based on a ***Partitioned*** **N**ext-**S**tate function. PINS connects language modules to analysis algorithms. The language modules compute for each specification a static dependency matrix, and implement a next-state function reflecting the operational semantics. The analysis algorithms access this abstraction of the specification, which still captures sufficient combinatorial structure to enable huge state space reductions. The key feature to this is the possibility to obtain transitions between subvectors. Due to lack of space, full details are provided elsewhere [3,4].

In a nutshell, a state for PINS is a vector of N slots, where a single slot can represent anything. The transition relation is split disjunctively into K groups. The $K \times N$ Boolean *dependency matrix* then denotes on which slots each group might depend. Dynamically, a dependency matrix is exploited as follows. Assume that transition group k depends on a short vector of state slots $\langle x_1, \ldots, x_\ell \rangle$ only. PINS next state function operates on this short vector, yielding a short next state, say $\langle y_1, \ldots, y_\ell \rangle$. Note that this result can be reused for many concrete states. By this single call we found a set of transitions on long state vectors: $\langle x_1, \ldots, x_\ell, a_{\ell+1}, \ldots, a_N \rangle \rightarrow \langle y_1, \ldots, y_\ell, a_{\ell+1}, \ldots, a_N \rangle$.

Finally, some optimizations can be expressed purely as transformations of the PINS matrix, also rewiring next-state calls. Such building blocks are implemented once, but all combinations of specification languages and analysis tools can benefit (Fig. 1).

The LTSMIN toolset consists of 28,000 lines of C Code.[2] The interfacing code for the supported frontends (DVE, NIPS, μCRL, mCRL2, our own ETF, Sec. 3.3) consists of only 200–500 lines each. The majority of code is in the three reachability tools, their support data structures, PINS2PINS wrappers, and the TORX [5] and CADP [1] connectors. Taken together, this yields 25 tool combinations, in addition to the minimization tool and various other support tools. The toolset is tested on Linux and MacOS X.

3 Functionality

3.1 Multiple Specification Languages

State-Based Languages. We implemented a language module for the DVE implementation of Barnat el al., giving access to the BEEM benchmark database [6]. Another language module connects the NIPSVM state generator [7], an interpreter for PROMELA, giving access to (pure) SPIN models [8]. The latter module could be refined by making the dependency matrix sparser for global variables and channels, which in general would improve the performance of the reachability tools.
Process Algebras. We have connected the native state generators of the μCRL [9] and mCRL2 [10] toolsets to LTSMIN. Both toolsets specify models in ACP-style process algebra with data, and are heavily used in industrial case studies [9]. They provide expressive ways to model systems, e.g., abstract data types (unbounded numbers, lists, trees), constrained data enumeration, and multi-way handshake communication.

Through the link with LTSMIN, users of all these tools gain for free 100% compatible enumerative, symbolic and distributed model checking tools, as well as compact state space storage formats and minimization tools.

3.2 Reachability and Minimization Tools

We implemented several tools for high-performance state space generation, in particular based on symbolic and distributed model checking. All exploration tools can check safety properties on-the-fly, and produce counter examples upon property violation. Alternatively, full state spaces can be generated and stored for minimization and analysis by external third-party model checkers.

Sequential: Implementations of standard enumerative reachability algorithms, using BFS or DFS search order. These PINS-based tools allow a base-line comparison with the native space generation facilities.

Symbolic: Implementations of symbolic reachability tools. Sets of states are stored as *(binary) decision diagrams*. The state space is computed symbolically by applications of the relational product. More precisely, for any specification language with an enumerative state generator implementing PINS, we automatically obtain a symbolic generator [3,4].

Distributed: Implementations of distributed state space generators, now based on the PINS interface, generalizing our earlier work [11]. This effectively combines the memory of many workstations, also achieving considerable speedups.

[2] Measured with David A. Wheeler's 'SLOCCount'.

PINS2PINS *wrappers:* All generators profit from optimizations in the PINS2PINS layer (Fig. 1). *Local transition caching* is useful for both enumerative generators; *tree compression* [11] is a technique for reducing memory footprint of enumerative generators; and *variable reordering* and *transition regrouping* [3] are useful for the symbolic generator, and in combination with transition caching.

Finally, in case of full state space generation, the LTSMIN toolset includes the distributed minimization tool `ltsmin-mpi` for (strong and branching) bisimulation reduction of labelled transition systems [12]. Also, Orzan's distributed τ-cycle elimination `ce-mpi` [13] tool is included. τ-Cycle freeness in turn admits the use of a simplified distributed minimization algorithm [14] for branching bisimulation. State based equivalences could be easily obtained by modifying the initial partition.

3.3 Tool Interoperability

Besides connecting to native state space generators of various languages (Sec. 3.1), LTSMIN provides converters or interfaces to third party back-end model checkers.

ETF. We defined our own Extended Table Format,[3] which enumerates all short transitions for all groups. It serves as input language of PINS, and as concise output format. E.g., we saw a 0.57 billion state LTS fit in a 1.6 Kb ETF file.

CADP and μCRL. LTSMIN has connections to the well-known CADP toolbox. State spaces can be exported in *binary coded graph* (BCG) format. LTSMIN also implements the CÆSAR/OPEN interface [1] to CADP's on-the-fly model checking and bisimulation algorithms. State spaces can be converted in μCRL's DIR format, allowing to use and compare against *their* implementation of distributed minimization tools.

DIVINE framework. The LTSMIN toolset includes a converter (`etf2dve`) from our ETF format to the input language of the DIVINE toolset [15], DVE. Thus, we obtain access to DIVINE's battery of distributed model checking algorithms. An interesting application is the certification of model checking results, to improve user confidence.

TORX testing tool. LTSMIN implements the TORX RPC interface (⟨*spec*⟩`2torx`), which allows *test case derivation* with TORX [5] for all PINS language modules. Additionally, JTORX allows checking two specifications for ioco-conformance [16].

GNA tool. In EC-MOAN,[4] the Genetic Network Analyzer [17] exports discrete abstractions of biological ODE models to ETF, and LTSMIN generates their state space for further analysis.

4 Experiments

We performed extensive benchmarking. Precise experimental results go beyond the scope of this tool paper. As illustration, the log-log scatter plot in Fig. 2 shows how distributed and symbolic model checking tools complement each other on selected DVE

[3] `http://fmt.cs.utwente.nl/tools/ltsmin/etf.html`

[4] European FP6 project on biological cell modeling and analysis, see `http://www.ec-moan.org/`

models from the *BEnchmarks for **Explicit** Model Checkers* (BEEM) database [6], ranging from 3×10^6 to 0.57×10^9 states. Each point represents two runs for one specification. The vertical axis indicates the wall-clock time (in seconds) for symbolic reachability (using variable reordering and the chaining heuristic); the horizontal axis denotes the time taken by distributed reachability (on 8×6 cores; with transition caching). The two models near the bottom-right corner are cases where symbolic methods are more than two orders of magnitude faster, whereas for `lift.[78]` and `pgm_protocol.8` the distributed tool is faster by more than factor 10. These are the first reported BDD-based experiments on benchmarks from the BEEM database, whose models are naturally biased towards enumerative methods.

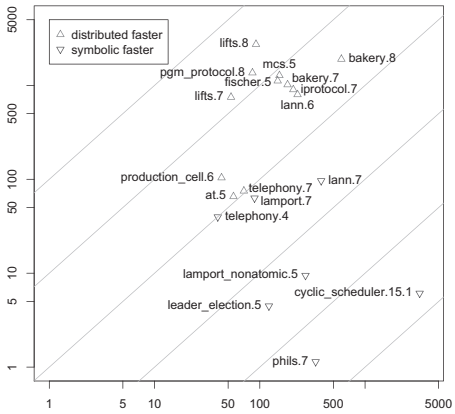

Fig. 2. Wall-clock time in seconds for distributed (x) and symbolic (y) reachability

In INESS,[5] LTSmin is used for the safety analysis of novel railway interlocking specifications. XUML statecharts are translated to MCRL2, and analyzed for safety properties by LTSMIN [18]. Depending on the track layout, we generated state spaces of up to 1.5×10^{11} states directly from MCRL2 models, by means of our symbolic tools.

References

1. Garavel, H.: OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
2. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
3. Blom, S.C.C., van de Pol, J., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, University of Twente, Enschede (2009)
4. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
5. Tretmans, G.J., Brinksma, H.: TorX: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 13–43 (2003)
6. 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)
7. Weber, M.: An embeddable virtual machine for state space generation. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 168–186. Springer, Heidelberg (2007)

[5] European FP7 project on INtegrated European Signalling Systems, `http://www.iness.eu/`

8. Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
9. Blom, S.C.C., Calamé, J.R., Lisser, B., Orzan, S., Pang, J., van de Pol, J., Dashti, M.T., Wijs, A.J.: Distributed analysis with μCRL: a compendium of case studies. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 683–689. Springer, Heidelberg (2007)
10. Groote, J., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., Weerdenburg, M.v., Wesselink, W., Willemse, T., Wulp, J.v.d.: mCRL2 toolset. In: Proc. of the IW on Advanced Software Development Tools and Techniques (2008)
11. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State-Space Generation. J. Logic Computation (2009) (to appear in print)
12. Blom, S., Orzan, S.: Distributed state space minimization. STTT 7(3), 280–291 (2005)
13. Orzan, S.: On distributed verification and verified distribution. PhD thesis, VU Amsterdam, The Netherlands (2004)
14. Blom, S., van de Pol, J.: Distributed branching bisimulation minimization by inductive signatures. In: Brim, L., van de Pol, J. (eds.) Proc. of 8th Parallel and Distributed Methods in verifiCation. ENTCS, vol. 14, pp. 32–46 (2009)
15. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
16. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
17. de Jong, H., Geiselmann, J., Hernandez, C., Page, M.: Genetic network analyzer: qualitative simulation of genetic regulatory networks. Bioinformatics 19(3), 336–344 (2003)
18. Hansen, H.H., Ketema, J., Luttik, S.P., Mousavi, M.R., van de Pol, J.C.: Towards model checking executable UML specifications in mCRL2. Innovations in Systems and Software Engineering 6(1-2), 83–90 (2010)