Symbolic Model-Checking Using ITS-Tools

Yann Thierry-Mieg

Sorbonne Universités, UPMC Univ. Paris 6, LIP6, and CNRS UMR 7606,
4 place Jussieu, F-75252 Paris Cedex 05, France
yann.thierry-mieg@lip6.fr

Abstract. We present verification toolset ITS-tools, featuring a symbolic model-checking back-end engine based on hierarchical set decision diagrams (SDD) that supports reachability, CTL and LTL model-checking and a user-friendly eclipse based front-end. Using model transformations to a Guarded Action Language (GAL) as intermediate format, ITS-tools can analyze third party (Uppaal, Spin, Divine...) specifications.

1 Introduction

ITS-tools is a symbolic model-checker relying on state of the art decision diagram (DD) technology. It offers model-checking (CTL, LTL) of large concurrent specifications expressed in a variety of formalisms: communicating process (Promela, DVE), timed specifications (Uppaal timed automata, time Petri nets) and high-level Petri nets. We are focused on verification of (large) globally asynchronous locally synchronous specifications, an area where DD naturally excel due to independent variations of (small) parts of the state signature.

We leverage model transformation technology to support model-checking of domain specific languages (DSL). Models are transformed to the Guarded Action Language (GAL), a simple yet expressive language with finite Kripke structure semantics.

Most of this paper is a discussion of the elements visible in Fig. 1. The top of the figure corresponds to the front-end (sections 2, 3), and is embedded in Eclipse, while the bottom of the figure corresponds to the back end (sections 4, 5).

2 Guarded Action Language

We define GAL as a pivot language that essentially describes a generator for a labeled finite Kripke structure using a C like syntax. This simple yet expressive language makes no assumptions on the existence of high-level concepts such as processes or channels. While direct modeling in GAL is possible (and a rich eclipse based editor is provided), the language is mainly intended to be the target of a model transformation from a (high-level) language closer to the end-users.

A **GAL** model contains a set of integer variables and fixed size integer arrays defining its state, and a set of guarded transitions bearing a label chosen from a finite set. We use C 32 bit signed integer semantics, with overflow effects; this ensures all variables have a finite (if large 2^{32}) domain. GAL offers a rich signature consisting of all C operators for manipulation of the `int` and `boolean` data type and of arrays (including nested

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 231–237, 2015.
DOI: 10.1007/978-3-662-46681-0_20

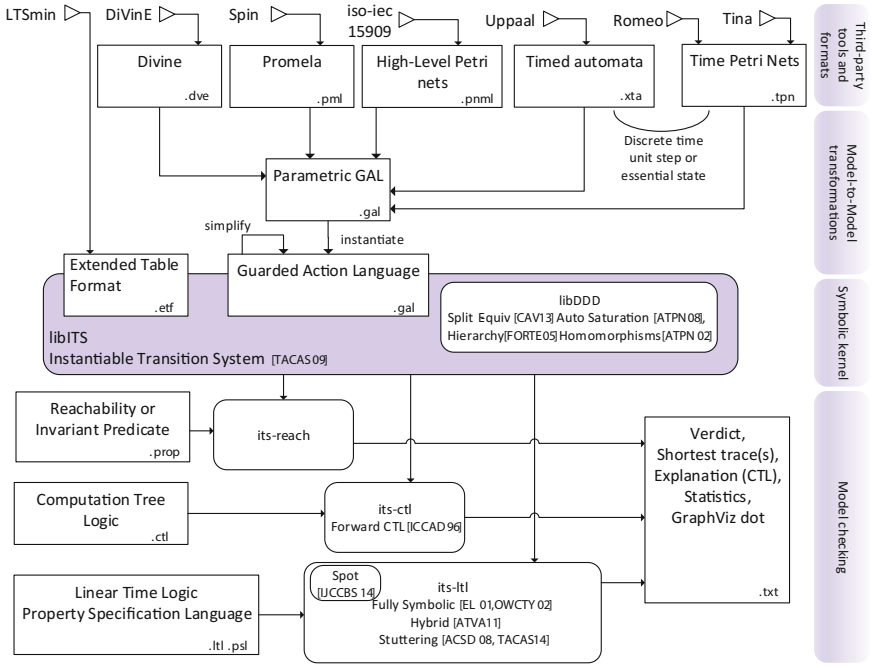

Fig. 1. Architecture of ITS-tools. Square boxes are files, rounded boxes are tools

array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. In any state (i.e. an assignment of values to the variables and array cells of the GAL) a transition whose boolean guard predicate is true can fire executing the statements of its body in a single atomic step. The body of the transition is a sequence of statements, assigning new values to variables using an arithmetic expression on current variable values. A special $call(\lambda)$ statement allows to execute the body of any transition bearing label λ, modeling non-determinism as a label based synchronization of behaviors. A special fixpoint instruction is provided allowing to express modal μ-calculus least and greatest fixpoints thus giving the language a potent expressive power.

Parametric GAL specifications may contain parameters, that are defined over a finite range. These parameters can be used in transition definitions, compactly representing similar alternatives. They can also be used to define finite iterations (for loop), and as symbolic constants where appropriate. Parameters do not increase expressive power, the symbolic kernel does not know about them, as specifications are instantiated before model-checking. The tool applies rewriting strategies on parametric transitions before instantiation, in many cases avoiding the polynomial blowup in size resulting from a naive parameter instantiation. Rewriting rules that perform static simplifications (constant identification...) of a GAL benefit all input formalisms.

Model to Model Transformations. Model-driven engineering (MDE) proposes to define domain specific languages (DSL), which contain a limited set of domain concepts [28]. This input is then transformed using model transformation technology to produce executable artifacts, tests, documentation or to perform specific validations. In this context GAL is designed as a convenient target formally expressing model semantics. We thus provide an EMF [1] compliant meta-model of GAL that can be used to leverage standard meta-modeling tools to write model to model transformations. This reduces the adoption cost of using formal validation as a step of the software engineering process.

3 Third-Party Support

We have implemented translations to GAL for several popular formalisms used by third party tools. We rely on XText for several of these: with this tool we define the grammar and meta-model of an existing formalisms, and it generates a rich code editor (context sensitive code completion, on the fly error detection,...) for the target language. The editor obtained after some customization is then often superior to that of the original tool. We applied this approach for the DVE language of DiVinE [5], the Promela language of Spin [3] and the Timed Automata of Uppaal [4] (in Uppaal's native XTA syntax).

The translation for DVE (succinctly presented in [13]) is quite direct, since the language has few syntactic constructs, and they are almost all covered by GAL. Channels are modeled as arrays, process give rise to a variable that reflects the state they are in. Similarly, the translation for Promela presents no real technical difficulty, although a first analysis of Promela code is necessary to build the underlying control flow graph (giving an automaton for each process). We currently do not support functions and the C fragment of Promela.

Discrete Time. The support for TA and TPN uses discrete time assumptions. Note that analysis in the discrete setting has been shown to be equivalent to analysis in a dense time setting provided all constraints in the automata are of the form $x \leq k$ but not $x < k$ [21,8]. For both of these formalisms, we build a transition that represents a one time unit delay and updates clocks appropriately. This transition is in fact a sequence of tests for each clock, checking if an urgent time constraint is reached (time cannot elapse), if the clock is active (increment its counter) or if it is inactive either because it will be reset before being read again, or because it has reached a value greater than any it could be tested against before a reset (do nothing).

A translation from high-level Petri nets (HLPN) conforming with the recent iso standard (thus produced by a variety of tools) is also available. HLPN are roughly to Place/-Transition nets what parametric GAL are to GAL: they are not more expressive (if all data types are finite) but they are much more compact and readable. Interestingly, the instantiation of GAL parameters is often much less explosive than the translation from HLPN to P/T nets: synchronizations of independent behaviors (e.g. interaction between a server S and a client C) can be represented using a sequence of $call(\lambda)$ in GAL, where the P/T net must explicitly have a transition for each possible synchronization choice.

4 Symbolic Kernel

ITS-tools use symbolic representations of sets of states using decision diagrams to face the combinatorial state space explosion of finite concurrent systems. Its kernel is **libDDD**, a C++ decision diagram library supporting Data Decision Diagrams (DDD [15]) and hierarchical Set Decision Diagrams (SDD [16]). Operations on these decision diagrams are encoded using homomorphisms [15], giving a user great flexibility and expressive power. The library can automatically and dynamically rewrite these operations to produce saturation effects in least fixpoint computations [20]. The Split-equiv algorithm introduced in [13] enables efficient evaluation of complex expressions including array subscripts and arithmetic, a feature heavily used to symbolically encode the semantics of GAL.

libITS is a C++ library built on top of libDDD, offering a simple and uniform API to write symbolic model checking algorithms for any system that can be described as an Instantiable Transition System (ITS). An ITS is essentially a labeled transition system with successor and predecessor functions described as operating on sets of states, and a boolean predicate function enabling state based logic reasoning. The tool supports compositions of labeled transition systems by directly using hierarchy in the state representation reflecting the composition [27]. libITS has native adapters for several formalisms (not represented on the figure), we focus in this paper on GAL.

ETF Support. A native ETF to ITS adapter is provided with libITS, supporting this output format of LTSmin. ETF files [10] represent the semantics of a finite Kripke structure in a format adapted to symbolic manipulation. This allows to analyze (CTL, LTL) models expressed in the many formalisms that LTSmin supports, provided generation of ETF succeeds (essentially if LTSmin can compute all reachable states).

5 Model-Checking

Using the ITS API we have built several model-checking tools. The tool **its-reach** can compute reachable states, and shortest witness paths (one or more if so desired) to target states designated by a boolean predicate. In a discrete time setting, this can be used to compute best or worst case time bounds on runs. It can also perform bounded depth exploration of a state space (a.k.a. bounded model-checking). It implements several heuristics to compute a static variable order for the input model.

The tool **its-ctl** performs verification of CTL properties (though fairness constraints are currently not supported). It reuses a component of VIS [11], a model-checking tool for verification and synthesis of gate level specifications, to transform input formulae into forward CTL form [22]. Forward CTL often allows (but not always) to use the forward transition relation alone, which is easier to compute than the backward (predecessor) transition relation. Hence forward CTL verification is more efficient in general, and furthermore many subproblems can be solved using least fixpoints (e.g. Forward Until) that benefit from automatic saturation at DD level.

The tool **its-ltl** performs hybrid (i.e. that build an explicit graph in which each node stores a set of states as a decision diagram) or fully symbolic verification of LTL and

PSL properties. The transformation of the formula into a (variant of) Büchi automaton and the emptiness checks of the product for hybrid approaches rely on Spot [24,17], a library for LTL and PSL model-checking. Fully symbolic model-checking uses forward variants of Emerson-Lei [19] or One-Way Catch Them Young [26]. The hybrid approaches efficiently exploit saturation and often outperform fully symbolic ones [18]. When the property is stuttering invariant (e.g. $LTL \setminus X$) we also offer optimized hybrid [23] and fully symbolic [7] algorithms that exploit saturation.

Other prototypes for solving games [29] and to exploit symmetries [14] on top of decision diagrams have been built, showing the versatility of the ITS API, but these tools are not part of the current release.

6 Case Studies and Experiments

In [6] ITS-tools were used to analyze compositions of time Petri nets produced from a DSL VeriSensor dedicated to wireless sensor network modeling. The specification analyzed contained around 50 clocks, many of which are concurrently enabled, preventing analysis by explicit tools such as Tina. With "its-reach" functional properties could be checked as well as quantitative measures such as worst-case lifetime analysis. In the Neoppod project [12] the CTL component was used to verify response and consistency properties of a protocol for a distributed database. Inria's Atsyra project [25] computes attack defense trees from a DSL using a model-to-model transformation to GAL.

In terms of raw benchmark power, ITS-tools participated in several editions of the model-checking contest at Petri nets conference, ranking first place in several categories [2]. It is compared favorably to LTSmin and to SAT solver Superprove on the benchmark BEEM[13]. It outperformed the symbolic tool Smart using its own benchmark models in [27]. On timed models, comparisons to Uppaal show that we tend to scale better in number of clocks, but are more sensitive to large bounds on clocks, something that was reported in previous similar experiments [9].

7 Conclusion

The ITS-tools are freely available from the webpage `http://ddd.lip6.fr`, offering easy access to efficient symbolic model-checking for a wide range of formalisms thanks to the general purpose Guarded Action Language.

Acknowledgements. The ITS-tools is the result of many years of collaborative development with both colleagues and students at LIP6, without whom this tool presentation would not be possible.

References

1. Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`
2. Model checking contest @ petri nets home page, `http://mcc.lip6.fr/`
3. Spin model checker home page, `http://spinroot.com/`

4. Uppaal home page, http://www.uppaal.org
5. Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., Weiser, J.: DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 863–868. Springer, Heidelberg (2013)
6. Ben Maïssa, Y., Kordon, F., Mouline, S., Thierry-Mieg, Y.: Modeling and Analyzing Wireless Sensor Networks with VeriSensor: an Integrated Workflow. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) VIII, 24–47 (2013)
7. Ben Salem, A.E., Duret-Lutz, A., Kordon, F., Thierry-Mieg, Y.: Symbolic model checking of stutter-invariant properties using generalized testing automata. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 440–454. Springer, Heidelberg (2014)
8. Beyer, D.: Improvements in BDD-based reachability analysis of timed automata. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, p. 318. Springer, Heidelberg (2001)
9. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A tool for BDD-based verification of real-time systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003)
10. Blom, S., van de Pol, J., Weber, M.: LTSMIN: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
11. Brayton, R.K., et al.: VIS: A System for Verification and Synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)
12. Choppy, C., Dedova, A., Evangelista, S., Hong, S., Klai, K., Petrucci, L.: The NEO protocol for large-scale distributed database systems: Modelling and initial verification. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 145–164. Springer, Heidelberg (2010)
13. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Towards distributed software model-checking using decision diagrams. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 830–845. Springer, Heidelberg (2013)
14. Colange, M., Kordon, F., Thierry-Mieg, Y., Baarir, S.: State Space Analysis using Symmetries on Decision Diagrams. In: Application of Concurrency to System Design (ACSD), pp. 164–172. IEEE Computer Society (2012)
15. Couvreur, J.M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.A.: Data decision diagrams for Petri net analysis. In: Application and Theory of Petri Nets (ICATPN), pp. 129–158 (2002)
16. Couvreur, J.M., Thierry-Mieg, Y.: Hierarchical decision diagrams to exploit model structure. In: Formal Techniques for Networked and Distributed Systems (FORTE), pp. 443–457 (2005)
17. Duret-Lutz, A.: LTL translation improvements in Spot 1.0. International Journal on Critical Computer-Based Systems 5(1/2), 31–54 (2014)
18. Duret-Lutz, A., Klai, K., Poitrenaud, D., Thierry-Mieg, Y.: Self-loop aggregation product — A new hybrid approach to on-the-fly LTL model checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 336–350. Springer, Heidelberg (2011)
19. Emerson, E.A., Lei, C.L.: Modalities for model checking: Branching time logic strikes back. Science of Computer Programming 8(3), 275–306 (1987)
20. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical Set Decision Diagrams and Automatic Saturation. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 211–230. Springer, Heidelberg (2008)
21. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)

22. Iwashita, H., Nakata, T., Hirose, F.: Ctl model checking based on forward state traversal. In: Computer-Aided Design (ICCAD). pp. 82–87. IEEE/ACM (1996)
23. Klai, K., Poitrenaud, D.: MC-SOG: An LTL model checker based on symbolic observation graphs. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 288–306. Springer, Heidelberg (2008)
24. Spot, L.R.D.E.: a library for LTL model-checking, http://spot.lip6.fr/
25. Pinchinat, S., Acher, M., Vojtisek, D.: Towards synthesis of attack trees for supporting computer-aided risk analysis. In: Workshop on Formal Methods in the Development of Software (co-located with SEFM) (2014)
26. Somenzi, F., Ravi, K., Bloem, R.: Analysis of symbolic SCC hull algorithms. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 88–105. Springer, Heidelberg (2002)
27. Thierry-Mieg, Y., Poitrenaud, D., Hamez, A., Kordon, F.: Hierarchical set decision diagrams and regular models. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 1–15. Springer, Heidelberg (2009)
28. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
29. Zhang, Y., Bérard, B., Kordon, F., Thierry-Mieg, Y.: Automated Controllability and Synthesis with Hierarchical Set Decision Diagrams. In: Workshop on Discrete Event Systems (WODES). pp. 291–296. IFAC/Elsevier, Berlin, Germany (September 2010)