

Symbolic Model-Checking using ITS-tools

Yann Thierry-Mieg¹

LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4 place Jussieu, F-75252 Paris Cedex 05, France
yann.thierry-mieg@lip6.fr

Abstract. We present the symbolic model-checking toolset ITS-tools. The model-checking back-end engine is based on hierarchical set decision diagrams (SDD) and supports reachability, CTL and LTL model-checking, using both classical and original algorithms. As front-end input language, we promote a Guarded Action Language (GAL), a simple yet expressive language for concurrency. Transformations from popular formalisms into GAL are provided enabling fully symbolic model-checking of third party (Uppaal, Spin, Divine...) specifications. The tool design allows to easily build your own transformation, leveraging tools from the meta-modeling community. The ITS-tools additionally come with a user friendly GUI embedded in Eclipse.

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 and tools to support model-checking of domain specific languages (DSL). Input models are first transformed to the Guarded Action Language (GAL), a simple yet expressive language with finite Kripke structure semantics.

2 Tool Features

Most of this section is a discussion of the elements visible in figure 1. The bottom of the figure corresponds to the back end (sections 2.1, 2.2) while the top of the figure corresponds to the front-end (sections 2.3, 2.4), and is embedded in Eclipse.

2.1 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 **lib-DDD**, a C++ decision diagram library supporting Data Decision Diagrams (DDD [16])

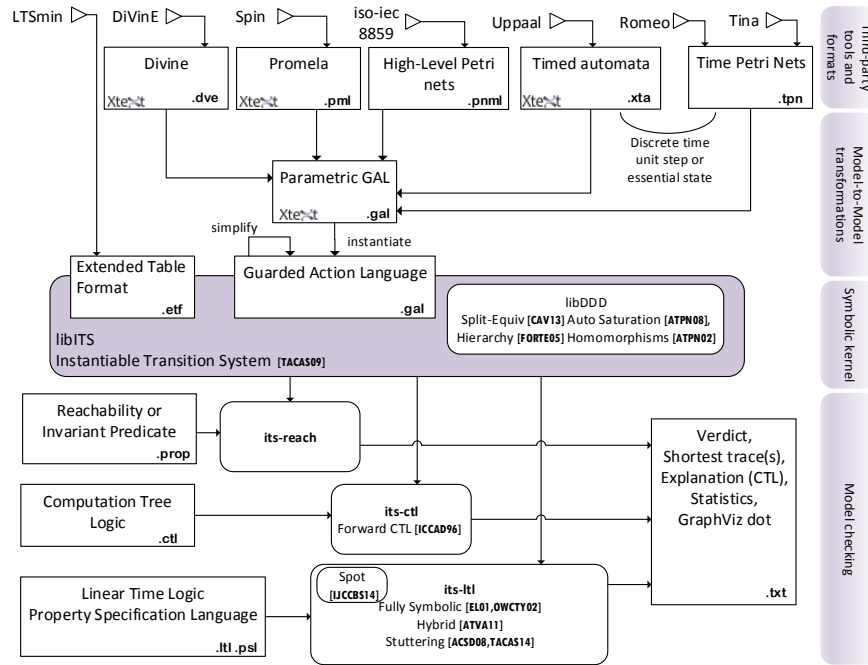


Fig. 1. Architecture of ITS-tools. Square boxes are files, rounded boxes are tools.

and hierarchical Set Decision Diagrams (SDD [17]). Operations on these decision diagrams are encoded using homomorphisms [16], 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 [21]. The Split-equiv algorithm introduced in [14] 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 [29]. 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 [11] 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).

2.2 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 [12] to transform input formulae into forward CTL form [23]. 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 [25, 18]. Fully symbolic model-checking uses forward variants of Emerson-Lei [20] or One-Way Catch Them Young [28]. The hybrid approaches efficiently exploit saturation and often outperform fully symbolic ones [19]. When the property is stuttering invariant (e.g. $LTL \setminus X$) we also offer optimized hybrid [24] and fully symbolic [7] algorithms that exploit saturation.

Other prototypes for solving games [31] and to exploit symmetries [15] 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.

2.3 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 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*(λ) 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 [30]. 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.

2.4 Third-party support

We have implemented translations to GAL for several popular formalisms used by third party tools. We rely on the eclipse project 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 (template proposals, type checking of expressions, scope...) 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 human readable XTA syntax).

The translation for DVE (succinctly presented in [14]) 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 $x < k$ [22, 9]. 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 transition is thus either a discrete change of automaton state or a delay of one time unit. With this semantics a trace with a shortest path will allow to count how many time steps are necessary to reach a certain goal. However, we also support a smaller abstraction of the state graph, that preserves CTL properties provided atomic properties of the formula do not refer to clocks. This abstraction called *essential states* was first defined [27] for time Petri nets in a discrete time setting (and is implemented notably by the tool Tina [8] see option $-SD$). It retains in the state graph only states that are reached through a discrete transition. While it abstracts sequences of time steps, it preserves location reachability and causality since all timings from a location are explored. Because discrete transitions frequently reset clocks, many (abstracted) states adjacent by a time step often lead to a single successor essential states, helping reduce the state graph size. This abstraction is implemented as a second transformation from the source TA or TPN that builds a GAL having essential state semantics. This second transformation highlights expressivity of GAL and the fine control it offers over atomicity of transitions, since we have to compute a least fixpoint (all states reachable by waiting will be abstracted) in each step of the transition relation.

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.

3 Case Studies and Experiments

In [6] it was 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 Neopod project [13] the CTL component was used to verify response and consistency properties of a protocol for a distributed database. Inria's Atsyra project [26] computes attack defense trees from a DSL using a model-to-model transformation to GAL.

In terms 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[14]. It outperformed the symbolic tool Smart using its own benchmark models in [29]. 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 [10].

4 Conclusion

The symbolic model-checker ITS-tools, its Eclipse based front-end, binaries for major platforms, source code as well as user documentation are freely available from the webpage <http://ddd.lip6.fr>. It offers easy access to efficient symbolic model-checking for a wide range of formalisms thanks to its support for 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. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV)*, LNCS 8044, pages 863–868. Springer, 2013.
6. Y. Ben Maïssa, F. Kordon, S. Mouline, and Y. Thierry-Mieg. 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. A. Ben Salem, A. Duret-Lutz, F. Kordon, and Y. Thierry-Mieg. Symbolic model checking of stutter invariant properties using generalized testing automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 8413, pages 440–454. Springer, 2014.
8. B. Berthomieu and F. Vernadat. Time petri nets analysis with tina. In *Quantitative Evaluation of Systems (QEST)*, pages 123–124. IEEE, 2006.
9. D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Formal Methods Europe (FME)*, LNCS 2021, pages 318–343. Springer-Verlag, 2001.
10. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for BDD-based verification of real-time systems. In *Computer Aided Verification (CAV)*, LNCS 2725, pages 122–125. Springer-Verlag, 2003.
11. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *Computer Aided Verification (CAV)*, pages 354–359. Springer, 2010.
12. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis.

- In *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.
13. C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO protocol for large-scale distributed database systems: Modelling and initial verification. In *Application and Theory of Petri Nets (ICATPN)*, LNCS 6128, pages 145–164. Springer, 2010.
 14. M. Colange, S. Baair, F. Kordon, and Y. Thierry-Mieg. Towards Distributed Software Model-Checking using Decision Diagrams. In *Computer Aided Verification (CAV)*, LNCS 8044, pages 830–845. Springer Verlag, 2013.
 15. M. Colange, F. Kordon, Y. Thierry-Mieg, and S. Baair. State Space Analysis using Symmetries on Decision Diagrams. In *Application of Concurrency to System Design (ACSD)*, pages 164–172. IEEE Computer Society, 2012.
 16. J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for Petri net analysis. *Application and Theory of Petri Nets (ICATPN)*, pages 129–158, 2002.
 17. J.-M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 443–457, 2005.
 18. A. Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, 2014.
 19. A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Self-loop aggregation product a new hybrid approach to on-the-fly LTL model checking. In *Automated Technology for Verification and Analysis (ATVA)*, pages 336–350. Springer, 2011.
 20. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, June 1987.
 21. A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical Set Decision Diagrams and Automatic Saturation. In *Applications and Theory of Petri Nets (ICATPN)*, LNCS 5062, 2008.
 22. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Automata, Languages and Programming*, pages 545–558. Springer, 1992.
 23. H. Iwashita, T. Nakata, and F. Hirose. Ctl model checking based on forward state traversal. In *Computer-Aided Design (ICCAD)*, pages 82–87. IEEE/ACM, 1996.
 24. K. Klai and D. Poitrenaud. MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs. In *Application and Theory of Petri Nets (ICATPN)*, LNCS, pages 288–306. Springer-Verlag, 2008.
 25. LRDE. Spot: a library for LTL model-checking. <http://spot.lip6.fr/>.
 26. S. Pinchinat, M. Acher, and D. Vojtisek. 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.
 27. L. Popova-Zeugmann and D. Schlatter. Analyzing paths in time petri nets. *Fundamenta Informaticae*, 37(3):311–327, 1999.
 28. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *Proc. of FMCAD'02 (FMCAD'02)*, volume 2517 of LNCS, pages 88–105. Springer.
 29. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 5505:1–15, 2009.
 30. M. Voelter, S. Benz, C. Dietrich, B. Engemann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
 31. Y. Zhang, B. Bérard, F. Kordon, and Y. Thierry-Mieg. Automated Controllability and Synthesis with Hierarchical Set Decision Diagrams. In *Workshop on Discrete Event Systems (WODES)*, pages 291–296, Berlin, Germany, Sept. 2010. IFAC/Elsevier.

Tool Demonstration : ITS-tools

All these demos can run on any platform. Before starting, you should download a recent Eclipse Luna, and point it at the update site <http://coloane.lip6.fr/night-updates>, then install the plugins of category “ITS Package”. The procedure is explained in more detail here:

<http://move.lip6.fr/software/DDD/itstools.php#sec:modinst>

These demos are focused on using the front-end and presenting GAL language. The actual command-line model-checking tools are not very flashy, but they could be shown as well to an interested audience. Similarly, there is no track focused on exhibiting meta-models and transformations, since this is considered a more technical topic, but the material to present these is available.

These demo scenarios are indicative, they are meant for about 20 minutes if done completely, but can be shortened to about 5 by burning some steps. At some point in the presentation the webpage <http://move.lip6.fr/software/DDD/gal.php> discussing syntax and semantics of GAL is shown.

Scenario 1: Verification of Uppaal specifications.

Description : A user wants to try the ITS tools to check an existing Uppaal specification. This scenario covers a presentation of the tools main components, though some of these parts can be skipped depending on the audience’s focus of interest. We use the Bridge and Vikings example of Uppaal distribution as running example.

Additional pre-requisite for step 1 : install Uppaal.

Step 1 (optional): Start the tool Uppaal. In Uppaal show the Bridge model (included in the demo/ folder of the distribution), a classic optimization problem about crossing a bridge that can only hold two “Vikings” in a minimal amount of time with only one lamp. End this step by clicking “save as...” and creating a .xta file. Note Uppaal does not have a built-in editor for these files, but they are native to it (command line Uppaal can read them).

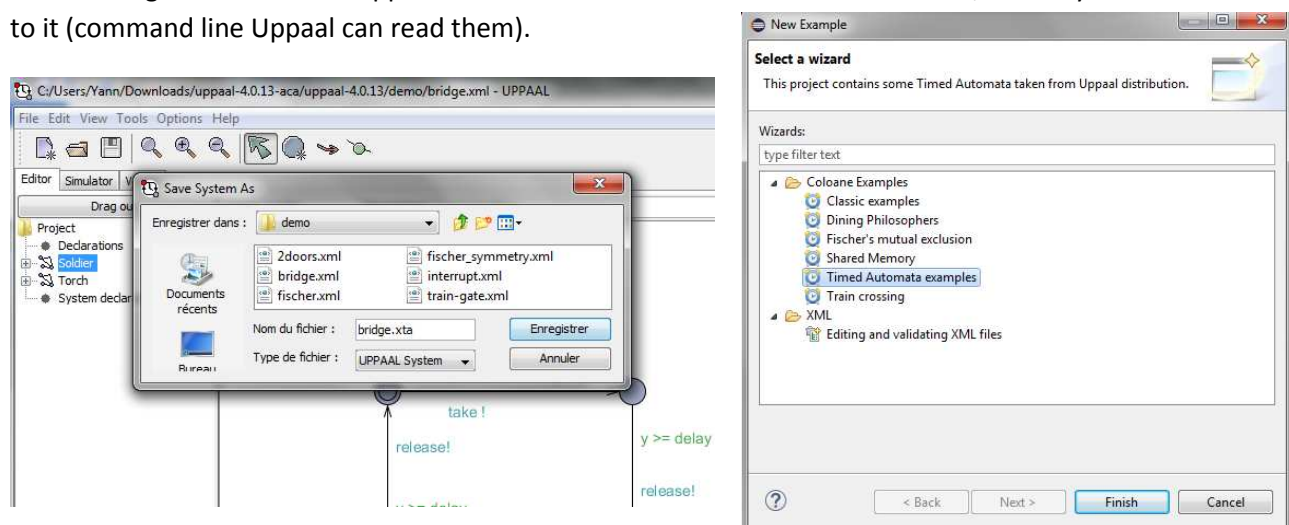


Figure 1 : Step 1 (save as xta in Uppaal) (left) and Step 2 (new example menu) (right) .

Step 2 : Start Eclipse. Highlight how a single click through the update site installs the tool : binaries for major platforms of the C++ components are embedded in the distribution. Use the menu “File-

>New->Example” and select “Timed automata examples”. Open the “Bridge.xta” by double clicking. It is the same file as that produced by Uppaal in step 1. Play around with the editor, show off on the fly error detection, code completion with “Ctrl-space” keyboard shortcut, syntax highlighting...

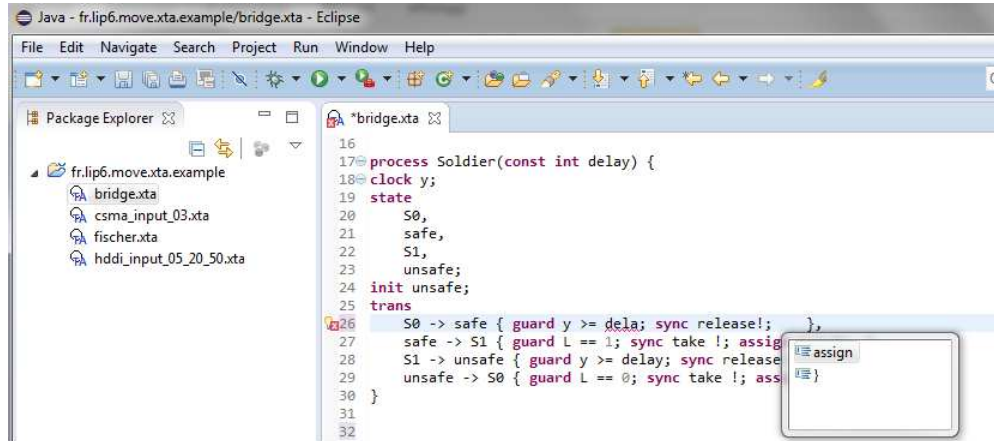


Figure 2 : Showing off the XTA editor, built using Xtext.

Step 3 : Transform to GAL. From the right-click context menu of the file bridge.xta, select action “TA to GAL->Transform to GAL (Time unit step). Variants of the translation can also be discussed depending on audience level of interest.

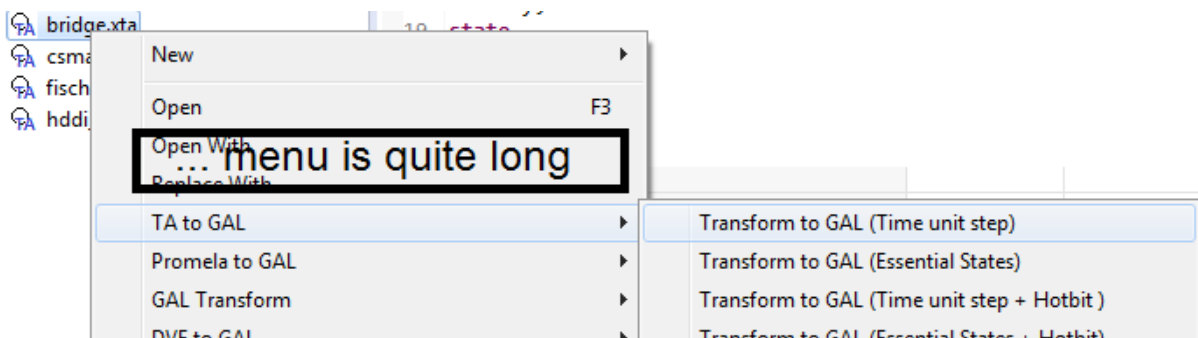


Figure 3 : translation.

Step 4 : Compare source and target model “bridge.one.gal”. We take some time here to introduce GAL syntax and explain its semantics. The translation makes heavy use of parameters, preserving much of the features of the Uppaal specification. Discuss the channel translation (calls in chantake and chanrelease), the transition “elapse”. Show a translation for a discrete transition of the TA. Show the two transitions “succ1” and “succ2” at end of file, that allow the system to progress through discrete transitions OR through a time step.

Step 5 : Open model “bridge.one.flat.gal”, resulting from parameter instantiation. Explain parameter instantiation. Notice that static simplifications allowed to simplify constants away, and to compute variable domains. This is a non-parametric GAL model we can verify properties on.

```

bridge.xta
34 state
35 one,
36 S0,
37 free,
38 two;
39 urgent;
40 S0;
41 init free;
42 trans
43 free -> S0 { sync take?; },
44 S0 -> one { },
45 S0 -> two { sync take?; },
46 one -> free { sync release?; as
47 two -> one { sync release?; };
48 }
49
50
51 const int fastest = 5;
52 const int fast = 10;
53 const int slow = 20;
54 const int slowest = 25;
55
56 Viking1 = Soldier(fastest);
57 Viking2 = Soldier(fast);
58 Viking3 = Soldier(slow);
59 Viking4 = Soldier(slowest);
60
61 system Viking1, Viking2, Viking3,
62
x-y bridge.one.gal
1 gal bridge ($fastest = 5, $fast = 10, $slow = 20, $slowest = 25)
2 typedef Torchid_t = 0 .. 3;
3 typedef Soldierid_t = 0 .. 3;
4 int glob_range_L = 0;
5 int glob_clock_time = 0;
6 /**Torch states [ 0=one, 1=S0, 2=free, 3=tw] */
7 array [1] Torch_state = (2);
8 /**Soldier states [ 0=S0, 1=safe, 2=S1, 3=unsaf]
9 array [4] Soldier_state = (3, 3, 3, 3);
10 array [4] Soldier_delay = ($fastest, $fast, $slow,
11 array [4] Soldier_clock_y = (0, 0, 0, 0);
12 /** To represent channel take */
13 transition chantake [true] label "dtrans" {
14 self."Sendtake";
15 self."Recvtake";
16 }
17 /** To represent channel release */
18 transition chanrelease [true] label "dtrans" {
19 self."Sendrelease";
20 self."Recvrelease";
21 }
22 transition elapse [true] label "elapseOne" {
23 for ($Torchid : Torchid_t) {
24 self."passUrgentTorch_$Torchid";
25 }
26 for ($Soldierid : Soldierid_t) {
27 self."updateClockSoldiery_$Soldierid";
28 }
29 }
x-y bridge.one.flat.gal
1 gal bridge_flat {
2 int glob_range_L = 0;
3 /**Torch states [ 0=one, 1=S0, 2=free, 3=tw] Dom
4 array [1] Torch_state = (2);
5 /**Soldier states [ 0=S0, 1=safe, 2=S1, 3=unsaf]
6 array [4] Soldier_state = (3, 3, 3, 3);
7 array [4] Soldier_clock_y = (0, 0, 0, 0);
8 /** To represent channel take */
9 transition chantake [true] label "dtrans" {
10 self."Sendtake";
11 self."Recvtake";
12 }
13 /** To represent channel release */
14 transition chanrelease [true] label "dtrans" {
15 self."Sendrelease";
16 self."Recvrelease";
17 }
18 transition elapse [true] label "elapseOne" {
19 self."passUrgentTorch_Torchid0";
20 self."updateClockSoldiery_Soldierid0";
21 self."updateClockSoldiery_Soldierid1";
22 self."updateClockSoldiery_Soldierid2";
23 self."updateClockSoldiery_Soldierid3";
24 }
25 /** State one is not urgent. */
26 transition passUrgentTorch_one_0 [Torch_state [0]
27 }
28 /** State free is not urgent. */
29 transition passUrgentTorch_free_0 [Torch_state [0]

```

Figure 4 : Steps 4 and 5 : Source XTA (left), parametric GAL (middle) and GAL (right).

Step 6 : Use Menu “File->New->Other->Coloane->ITS Composition Model” to create an invocation file. Open this file with the dedicated editor (double-click). Drag and drop the “bridge.one.flat.gal” file into the left part of the window (“Types editor”). Select the model, then in the left part click “Analysis”. Select “ITS Reachability” then write “Soldier_state[0]==1&&Soldier_state[1]==1&&Soldier_state[2]==1&&Soldier_state[3]==1” in the “Reachable ?” field and click “Run Service”. Open the ITS reachability section to access the run results. In the “Check Results Description” zone examine the “Raw output” field, this is a trace of the background its-tools invocation. This output contains a shortest trace (length 78 steps) that solves the minimization problem for the bridge example : one can count the number of “elapse” steps in the solution.

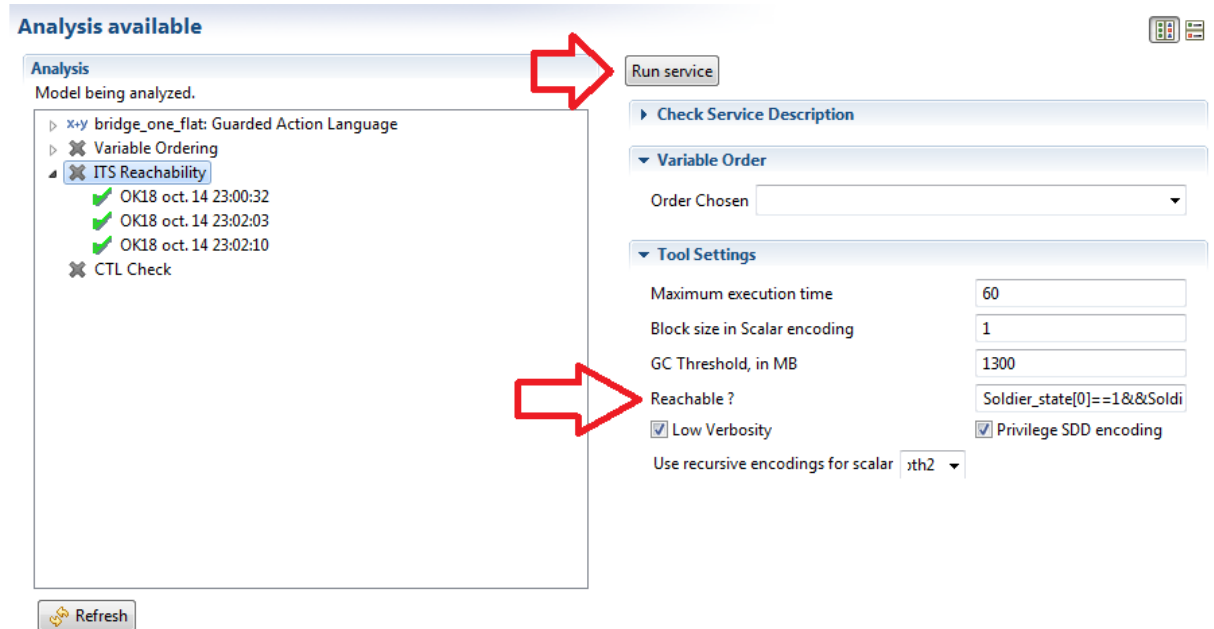


Figure 5 : Step 6 : invoking its-tools using the eclipse wrapper.

Scenario 2: Verification of Promela specifications.

Description : A user wants to try the ITS tools to check an existing Promela specification. This scenario is similar to the scenario 1, but uses a Promela model as running example.

Step 1 : Similarly to Step 2 of scenario 1, use menu “File->New->Example->Promela examples” to get started. Open the “hanoi.1.pml”. Highlight editor features; the Promela editor comes with above average type checking and template code completion, as well as a readable “Outline” view

Step 2 : Similarly to step 3 of Scenario 1, right-click the “.pml” file and select action “Promela to GAL->Transform to GAL”. Go through steps 4 and 5 of scenario 1, although the example is not as rich in terms of features of GAL used : this translation does not heavily use parameters. Explain how the process counter “pcVar” represents the state of a Promela process. We can still discuss the differences between “hanoi.gal” and the instantiated “hanoi.flat.gal” albeit less effectively than in step 5 of scenario 1.

Step 3 : Analysis, similarly to step 6 of scenario 1. We check for reachability property “ $b[1]==8 \wedge b[2]==7 \wedge b[3]==6 \wedge b[4]==5 \wedge b[5]==4 \wedge b[6]==3 \wedge b[7]==2 \wedge b[8]==1$ ”. This produces a trace solving the Hanoi towers problem.

Alternatively, using the model derived from “phils.8.pml”, in the CTL check tool, make sure “produce witness trace” is checked, “Add a formula” and use “DEADLOCK;” in the formula field. This produces a trace to deadlocks in the dining philosophers problem.

Scenario 3: Discussing GAL semantics and features.

Description : Our user is more interested in writing a transformation to GAL than using one of the existing ones. This demonstration focuses on features of GAL language. We use a Time Petri Net running example for its simplicity, or other small GAL examples from the web page. We suppose the audience knows the semantics of time Petri nets.

Step 1 : Create a simple time Petri net. Use the menu “File->New->Project->General->empty project” to create a project then “File->New->Other->Coloane->Model->Time Petri Net”. Using the palette, add two places “a” (with initial marking 1) and “b”, a transition “t”, and arcs to carry a token from a to b. Set the transition’s earliest and latest firing times to 3 and 5 respectively. You may need to use “Window->show view->other->properties” to edit the names and values of annotations in the net. The graphical editor is pretty nice, building a simple model is just a few clicks. Alternatively, use “New->Example->Coloane->Train” and use “train.model” as running example.

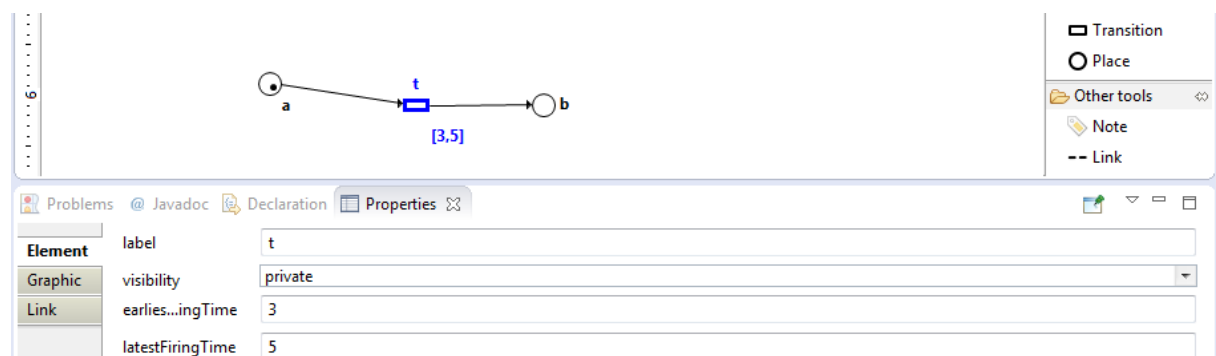
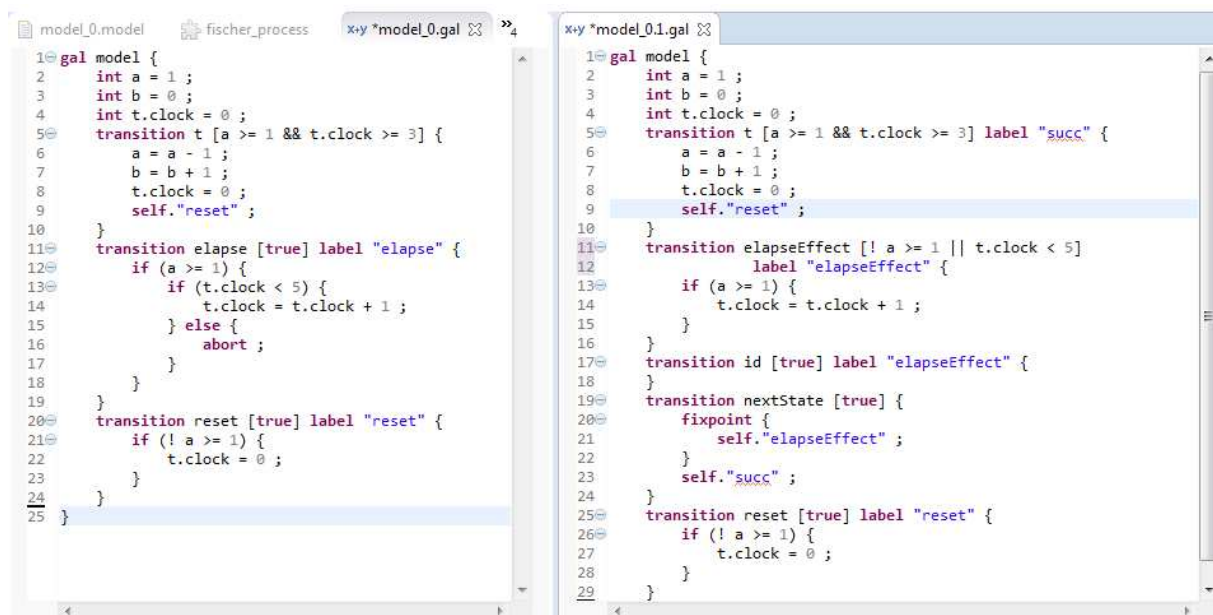


Figure 6 : Editing a TPN

Step 2 : Translate to GAL. Right-click the model then select “Export->Coloane->GAL file”. Examine the GAL model, since it is relatively simple the semantics of GAL are easy to understand. Now run the export again using the “Essential states” selection. Discuss the role of the fixpoint in this variant of a step of the transition relation : using the empty “identity” transition, and appropriate labels, the transition relation consists in a least fixpoint using “elapse” followed by firing of any discrete transition. We can run reachability analysis to compare state space sizes though this is more interesting in presence of more clocks or when increasing latest firing time significantly, where the reduction starts to shine.



```
1 gal model {
2   int a = 1 ;
3   int b = 0 ;
4   int t.clock = 0 ;
5   transition t [a >= 1 && t.clock >= 3] {
6     a = a - 1 ;
7     b = b + 1 ;
8     t.clock = 0 ;
9     self."reset" ;
10  }
11  transition elapse [true] label "elapse" {
12    if (a >= 1) {
13      if (t.clock < 5) {
14        t.clock = t.clock + 1 ;
15      } else {
16        abort ;
17      }
18    }
19  }
20  transition reset [true] label "reset" {
21    if (! a >= 1) {
22      t.clock = 0 ;
23    }
24  }
25 }
```

```
1 gal model {
2   int a = 1 ;
3   int b = 0 ;
4   int t.clock = 0 ;
5   transition t [a >= 1 && t.clock >= 3] label "succ" {
6     a = a - 1 ;
7     b = b + 1 ;
8     t.clock = 0 ;
9     self."reset" ;
10  }
11  transition elapseEffect [! a >= 1 || t.clock < 5]
12    label "elapseEffect" {
13    if (a >= 1) {
14      t.clock = t.clock + 1 ;
15    }
16  }
17  transition id [true] label "elapseEffect" {
18  }
19  transition nextState [true] {
20    fixpoint {
21      self."elapseEffect" ;
22    }
23    self."succ" ;
24  }
25  transition reset [true] label "reset" {
26    if (! a >= 1) {
27      t.clock = 0 ;
28    }
29  }
}
```

Figure 7 : Use of GAL fixpoint in discrete time semantics

Step 3 : Parameter instantiation. Build a “File->New->File” and name it “param.gal”. Copy or type (nice editor) this simple program :

```
gal test {
  int x = 0;
  int y = 0;
  typedef range = 1..3;

  transition t (range $x, range $y) [$x!=2] {
    x = $x;
    y = $y;
  }
}
```

Open right-click context menu on file param.gal , and run actions in category “GAL Transform”

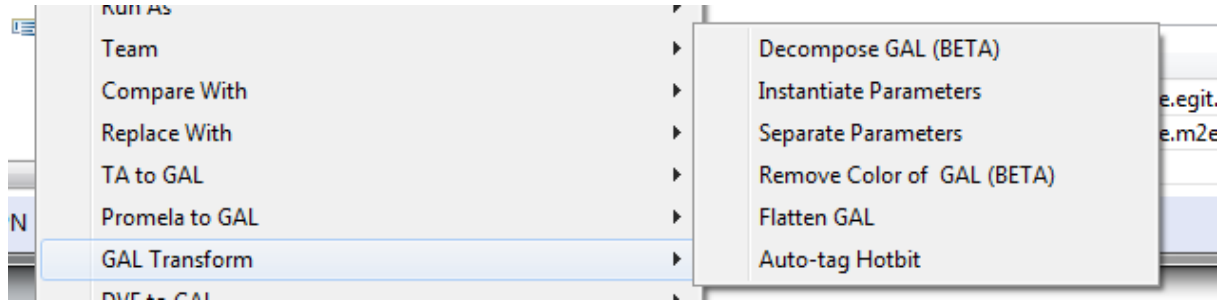


Figure 8 : GAL Transform menu

We will use “Separate parameters” first to explain the algorithm for parameter separation. This intermediate version “param.sep.gal” preserves parameters but rewrites transitions in an attempt to reduce the number of parameters on transitions. It is an intermediate product there for explanation : “Flatten” is the preferred method, that chains “separate” and “instantiate” actions with structural simplifications to produce a GAL adapted to model-checking. Run “instantiate” on the original “param.gal” and compare size of “param.inst.gal” model with that obtained with “flatten”. The quadratic explosion is avoided. We can play with variants on the model, for instance change the guard to “ $x \neq y$ ” and run the same experiments.

```

1 gal test_inst {
2   /** Dom:[0, 1, 3] */
3   int x = 0 ;
4   /** Dom:[0, 1, 2, 3] */
5   int y = 0 ;
6   transition t_1_1 [true] {
7     x = 1 ;
8     y = 1 ;
9   }
10  transition t_1_2 [true] {
11    x = 1 ;
12    y = 2 ;
13  }
14  transition t_1_3 [true] {
15    x = 1 ;
16    y = 3 ;
17  }
18  transition t_3_1 [true] {
19    x = 3 ;
20    y = 1 ;
21  }
22  transition t_3_2 [true] {
23    x = 3 ;
24    y = 2 ;
25  }
26  transition t_3_3 [true] {
27    x = 3 ;
28    y = 3 ;
29  }
}

1 gal test_flat {
2   /** Dom:[0, 1, 3] */
3   int x = 0 ;
4   /** Dom:[0, 1, 2, 3] */
5   int y = 0 ;
6   transition t [true] {
7     self."ty" ;
8     self."tx" ;
9   }
10  transition tx_1 [true] label "tx" {
11    x = 1 ;
12  }
13  transition tx_3 [true] label "tx" {
14    x = 3 ;
15  }
16  transition ty_1 [true] label "ty" {
17    y = 1 ;
18  }
19  transition ty_2 [true] label "ty" {
20    y = 2 ;
21  }
22  transition ty_3 [true] label "ty" {
23    y = 3 ;
24  }
25  }
}

```

Figure 9: Contrasting naïve instantiation (left) with parameter separation (right).

Notice in all these examples that the GAL encoding is very close to the symbolic encoding, hence smaller GAL specifications really are easier to model-check with DD, besides removal of the barrier due to handling polynomially larger specifications at every step of the workflow. These transformations are possible because we know about fine grain semantics of GAL actions, this contrasts with the opacity of LTSmin PINS API, a project offering several similar feature to GAL.