Statistical Model Checking in Uppaal: Lets Practice [⋆]

Alexandre David[1], Kim Guldstrand Larsen[1], Axel Legay[12], and Marius Mikučionis[1]

[1] Aalborg University, Denmark
[2] INRIA/IRISA Rennes, France

Abstract. Statistical model-checking is a recent technique used for both verification and performance analysis of hybrid systems. It does not suffer from decidability issues or state-space explosion compared to traditional model-checking. Furthermore, it is applicable to more powerful formalisms such as stochastic hybrid automata. Its principle is simple so how simple is it really to make it work in practice? In this extended abstract, we raise a number of practical issue, some of them disconnected from the underlying theory, and show how they are addressed in Uppaal SMC.

1 Introduction

Statistical model-checking (SMC) [14, 17, 18, 13] is an approach recently introduced as both a validation technique to reason about correctness of systems and an analysis technique to study performance. The idea behind SMC is to generate bounded runs of a system and to analyze the outcomes of these runs w.r.t. some linear temporal logic formula (or some variant of it). Statistical methods are then used to derive how many runs are necessary to assert with some given degree of confidence if the formula holds. This has several advantages over traditional model-checking. Computing the runs, even for a hybrid system, does not involve manipulating structures to handle symbolic representations of states such as BDD [6] or zones [4]. In addition, states are not stored and no special technique to represent the state-space is required. Lastly it is far easier to parallelize or distribute SMC [7] compared to traditional model-checking, e.g., no distributed hash table is necessary. The technique is simpler, cheaper, and scales well w.r.t. the size of the models and the actual hardware on which it runs.

SMC has a more powerful formalism and can actually work in practice on large systems without resorting to advanced abstraction techniques [9]. It also

[⋆] The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n 318003 (TREsPASS). This publication reflects only the authors views and the Union is not liable for any use that may be made of the information contained herein. Also the work was supported by the Danish VKR Center of Excellence MT-LAB (www.mt-lab.dk), and the Sino-Danish Basic Research Center IDEA4CPS (www.idea4cps.dk).

allows to analyze properties [8, 2] that cannot be expressed or even checked with classical model-checking. As a consequence, the technique is spreading to various research areas such as system biology [10, 12] or software engineering [19, 15].

In this paper we are interested in showing how SMC works in practice in UPPAAL SMC. Even if the technique is simpler than model-checking, it still presents its own challenges. If we compare the formalism used in traditional model-checking of timed systems, timed automata [1] or timed variants of Petri nets [16, 5] are used. UPPAAL SMC can handle stochastic hybrid automata without being plagued by decidability issues. On the other hand, the generation of runs must use numerical methods to integrate on-the-fly differential equations. The first problem is precision. A fined grained integration step affects speed negatively. The second problem is correctness because this interacts with the detection of the states we are looking for (through monitors or formulas). If we compare w.r.t. scalability, SMC faces extra challenges when generating the runs. In addition to estimating when to take transitions, which model-checking needs to do as well, SMC has to deal with distributions and has to resolve races between components. This also interferes with the numerical integration.

First we show some noteworthy technical implementation issues. Then we take two major points separately, namely how the computation of successors using our stochastic semantics works together with the integration, and scalability of the models.

2 Technical Issues

The general architecture of UPPAAL is based on a pipeline [3]. To keep the code uniform and ease the integration of a new engine, SMC follows the same principle. The different engines follow this structure:

– A main function that initializes the checker (hypothesis testing, probability estimation, etc . . .), generates runs, and analyzes the results.
– A function that i) estimates which delay each component of the model can take and ii) resolves which component should move.
– A function that decides for a given component and delay which transition is taken and takes it.
– A function to monitor/evaluates if a formula holds.

Now let us dig out the issues.

Checkers. The different checkers, though similar, work very differently. Hypothesis testing is using a sequential method with parameters that are different from the ones used for probability evaluation. From version 4.1.15, probability estimation is also using a sequential method, though a different one. The challenge is to fit very different algorithms and parameters into the same framework to reuse common functionalities. We use a different main function for each checker and that function reuses the same delay/successor functions. Second, we broadcast the parameters as $< key, value >$. The functions that need a certain value listen on this one, e.g., $< \alpha, 0.95 >$.

When Do We Stop? A run can be stopped if it reaches its bound, that is either a time bound, a step bound, or an arbitrary cost bound. Otherwise, a run stops prematurely if a certain state is detected. Stopping on the right bound means that any function computing a delay must be aware of this bound. In addition, reaching this bound must be signaled to the main loop to terminate the run. The detection of state also interferes with delays. Indeed, if the user wants to detect a state with a condition on a clock such as $3 \leq x \leq 5$, the checker should not delay 4 if $x = 2$ otherwise the state will not be detected. So any function computing a delay must also be aware of such bounds. Now comes the question: How to detect these bounds? The formulas may not be that simple and the bounds may be known only at run-time.

Concrete or Symbolic States? How to compute bounds on-the-fly with a concrete state? One way would be to use a symbolic state based on difference bound matrices (DBM) and compute a trajectory from a concrete clock valuation. We would lose all benefits of *not* having symbolic states. So we use a cheap version with the *decorator* pattern: We wrap our concrete state inside a decorator state that intercepts all calls that evaluate clock constraints. It evaluates delays from the current valuation to satisfy or stay within some constraint. This in turn computes delay intervals. These intervals are used to detect when a formula is satisfied but also when a guard on a transition holds (and that is used to pick the actual delay). Not all issues on complex formulas are currently dealt with but the solution is lightweight and sound.

Precision. Floating-point numbers are only approximations. Any computation using them is wrong up to some ϵ. It does not make sense to test for equality for example. Lets go back to our hybrid model that takes root in timed automata. What do we do with $x \geq 3$, $x > 4$, or $x == 3$? This is taking into account that x is not exact. We have to do this up to some ϵ value that is for the moment a constant in the code. The case $x == 3$ is interesting because having it on a guard with a delay up to, say, 4 gives probability 0 to take the transition. But having it on the same transition with a delay bounded by 3 because of an invariant gives probability 1 to take it at 3 (which is possible in case of, e.g., time-out modeling). The current solution is to stick the clock valuation to exact integer valuations when they get “too close” to integer points (up to some ϵ constant). The rationale behind this comes from the probability of getting to such a point in the first place. It is 0 for delays picked according to some distribution but 1 if the delay is forcibly bounded by an integer.

UPPAAL *Features.* Urgent states or transitions are allowed in UPPAAL. These are special cases where delays are not allowed. Invariants in timed automata have the *feature* to effectively disable transitions if a certain delay would violate the invariant of the target state. For example going from A to B with $x \leq 5$ on B is not possible from A if $x == 6$, *even if the guard allows this* (and the x is not reset). This is forbidden in SMC and the system will be declared not sane because this interferes with the stochastic semantics (problem with the distributions, in particular exponential) and it would make the implementation needlessly complex.

Memory Consumption. The main idea behind SMC is to generate runs on-the-fly. The engine generates successor states and does not need to remember previous ones. This is a major advantage over model-checking. So when do we need memory? If a user wants to analyze visually the results, a plot is needed. The plot comes from data generated on-the-fly. Now it is hopeless to store data of 1000 of runs if some integration step of, say 10^{-4}, is used on runs bounded by 1000 time units. These are not outrageous number, but multiplied (by the size of the datum to be stored as well), they are. In fact, we have this issue with Matlab-Simulink. Instead, Uppaal SMC filters the data on-the-fly according to some resolution used for plotting.

MPI. The distributed version of SMC is using the message passing interface (MPI) library. Both hypothesis testing and probability estimation are supported. The first issue here is theoretical: There may be a bias in the result if some computing nodes deliver their outcomes faster than others. A balancing algorithm for the result [7] must be implemented. The second issue is practical: A distributed termination algorithm must be implemented on top to decide on-the-fly when to terminate.

3 Numerical Integration and Stochastic Semantics

Briefly, when a component can take a transition, a delay is picked according to a distribution. If the current state has an invariant, the distribution is uniform. The lower bound is given by the guard on the transition and the upper bound by the invariant. Guard should not have upper bounds but in the case of uniform distributions, this will also work. If the current state does not have an invariant, then the delay is picked according to an exponential distribution whose rate must be given. Now what if a clock rate is some (arbitrary) expression and not simply 0 or 1?

The way this problem is solved is to consider another component in parallel with the system that wants to move in δ time units. It simply races with the rest of the system with all the rates of the clocks being constant during that amount of time. Most of the time this component will win the race and then it updates the rates of all clocks, after which the race resumes. When the system, or even better the current state, is not using such rates, the delays may be taken in big steps. Otherwise we have to integrate. The current method used is Euler's algorithm for simplicity but we will investigate more efficient ones such as Runge-Kutta's. The point is that we can still keep this *semantically* as another separated component.

The numerical integration interferes with the global time bound on runs and the evaluation of invariants (or maximum delays). As an additional difficulty, invariants of the type $x \geq -1$ now make sense and may result in unbounded delay if $x' \geq 0$. This affects the distribution dynamically.

Another orthogonal issue, which we unfortunately have not solved at the moment, is numerical stability. If the δ step is taken too large or too small, the integration will not work. Wrong results or *Zeno* behaviour will be detected[3]. Currently if a run has more than 100000 states consecutively in 0.3 time units or less, then the engine considers that the run is not making time progress and declares it as Zeno. We need some heuristic to stop the engine in case of real Zeno behaviour.

4 Scalability Issues

How to keep up performance even when the number of component increases? The fundamental step in generating runs is to compute a successor, and before that the delay for taking a transition. The distribution of the delay is affected by the guards on the transitions. This means that every component has to evaluate every guard of every outgoing transition in its current state to figure out the distribution and which one to take.

The problem is that only one component out of potentially many moves, so all the computation from all the other components are wasted. The solution is to remember the choices of all components to avoid to recompute them when it is possible. A simple update that reflects the delay taken is much cheaper than re-evaluating the delay from scratch. So when is it possible? If a component moves independently, which means no side-effect on guards, rates, or invariants, then we can do it. In practice, Uppaal SMC consider any use of global variable because its static analyzer is not powerful enough. There are also different solutions to keep track of dependency between transitions that incur different memory overheads, e.g., state-based, or transition-based, or expression-based. Obviously (combinatorial blow-up) we want to limit the amount of pre-computations and use techniques such as bit-vector operations to efficient set operations, e.g., to detect if a written x is also read by another component.

The issue of independent moves between components is difficult to implement but it has a major performance impact.

5 Conclusion

We have shown a number of technical challenges that are faced with statistical model-checking. They are mostly of numerical nature, which is not an issue for traditional model-checking because of the symbolic nature of the algorithms employed. There are still improvements to be done on the numerical solver or static analyzer of Uppaal that are purely technical. Other more fundamental improvements are on so called *rare events*. Our current technique requires huge amount of runs (billions or more) if we want good confidence that some state is not reachable, except that it happens rarely.

[3] Whenever a run has an infinite number of discrete transitions in a finite amount of time, it is Zeno.

The SMC technique is not bullet proof but nicely complement model-checking, in particular for finding bugs. Over-approximation techniques can safely declare that bad states are not reachable. SMC can safely declare that they are reachable, up to numerical errors. Numerical issues are starting to be taken into account [11].

SMC is a promising technique primarily because it bypasses decidability issues faced by model-checking but it does have its own challenges compared to model-checking and it is certainly not trivial to do it right.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010.
3. G. Behrmann, A. David, K. G. Larsen, and W. Yi. A tool architecture for the next generation of UPPAAL. In *UNU/IIST 10th Anniversary Colloquium. Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 352–366, 2003.
4. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.
5. F. Bowden. A brief survey and synthesis of the roles of time in petri nets. *Mathematical and Computer Modelling*, 31(1012):55 – 68, 2000.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
7. P. Bulychev, A. David, K. G. Larsen, A. Legay, and M. Mikucionis. Distributed parametric and statistical model checking. In K. H. Jiří Barnat, editor, *Proceedings 10th International Workshop on Parallel and Distributed Methods in Verification*, volume 72 of *EPTCS*, pages 30–42.
8. E. Clarke, A. Donzé, and A. Legay. On simulation-based probabilistic model checking of mixed-analog circuits. *Formal Methods in System Design*, 36:97–113, 2010.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
10. H. Gong, P. Zuliani, A. Komuravelli, J. R. Faeder, and E. M. Clarke. Computational modeling and verification of signaling pathways in cancer. In *ANB*, volume 6479 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2010.
11. A. E. Goodloe, C. Muoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 441–446, 2013.
12. S. K. Jha, E. M. Clarke, C. J. Langmead, A. Legay, A. Platzer, and P. Zuliani. A bayesian approach to model checking biological systems. In *CMSB*, volume 5688 of *LNCS*, pages 218–234. Springer, 2009.
13. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Perform. Eval.*, 68(2):90–104, 2011.

14. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.
15. J. Martins, A. Platzer, and J. Leite. Statistical model checking for distributed probabilistic-control hybrid automata with smart grid applications. In *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2011.
16. W. Penczek and A. Pólrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*, volume 20 of *Studies in Computational Intelligence*. Springer, 2006.
17. K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114, pages 202–215. Springer, 2004.
18. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, LNCS 2404, pages 223–235. Springer, 2002.
19. P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In *HSCC*, pages 243–252. ACM, 2010.