Self-Adaptive Model Checking, the next step?

Fabrice Kordon[1] and Yann Thierry-Mieg[1]

Sorbonne Université, CNRS LIP6, F-75005 Paris, France
Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr

Abstract. Model checking is becoming a popular verification method that still suffers from combinatorial explosion when used on large industrial systems. Currently, experts can, in some cases, overcome this complexity by selecting appropriate modeling and verification techniques, as well as an adapted representation of the system. Unfortunately, this cannot yet be done automatically, thus hindering the use of model checking in industry.
The objective of this paper is to sketch a way to tackle this problem by introducing *self-adaptive model checking*. This is a long term goal that could lead the community to elaborate a new generation of model checkers able to successfully push forwards the scale of the systems they can deal with.

Keywords: Verification, Model checking, Formal methods and methodology, benchmark for verification.

1 Introduction

Model checking is becoming a popular verification method, even in large companies such as Intel, Motorola and IBM [18]. There are already many success stories involving this technique. PolyORB, an open source middleware now used in aerospace applications, was formally verified to prove that no deadlocks nor livelocks could occur on one execution node [24]. Similarly, some aspects of the bluetooth interaction protocols have been studied formally [12]. Finally, NASA development of critical code also involved model checking to ensure its safety [23].

The main advantage of model checking is to be quite easy to automate, and so, it can be operated by non experts. Unfortunately, it suffers from the so-called combinatorial state explosion, that is difficult to tackle, especially for non experts. This can be called the "model checking dilemma" : on the one hand, this approach is easy to use but as soon as you deal with complex problems, an expert aware of the various appropriate techniques and algorithms is required to complete the verification task.

Today, numerous techniques have been defined by the various communities working on the topic. They may rely on several types of automata like Büchi [6], Rabin [30], Streett [34], testing automata [17] and variants [2], etc. They may also involve several techniques to describe the system such as symmetry reductions [8], various types of decision diagrams [7,33,20], partial order reductions [10], on-the-fly automata reductions [15], etc. Moreover, sometimes, several techniques are combined like Binary decision diagrams and symmetry reductions in [36], on-the-fly reductions and hierarchical

decision diagrams in [13], and hierarchical decision diagrams and symmetry reductions in [11]. Finally, the analysis of properties is also a part of the optimization problem since there may exist some particular cases where some adapted algorithm performs better [31], automata derived from formulas can also be optimized [1] and observed elements in the system taken into consideration to reduce complexity [26].

However, when analyzing the behavior of model checkers on large benchmarks, such as the one of the Model Checking Contest[1] [28], we can notice that several combinations of techniques can be successfully operated to solve some classes of problems (e.g. LTL, CTL, reachability, bound computation, etc.). Identifying such combinations of techniques is thus of great help.

But so far, only experts can estimate which combination of techniques will be the more likely to solve a complex verification problem. The idea of *"self-adaptive"* model checking is to define the bases of an infrastructure embedding the capability to select and use the best data-representations and algorithms so that it can transparently tackle the complexity when performing verification.

This paper is structured as follows. Section 2 presents a simple analysis of the techniques used by model checking tools participating to the Model Checking Contest in 2015, 2016 and 2017. In particular, we are trying to extract some information about the involved techniques and focus on determining wether symbolic (based on decision diagram) or explicit approaches are the most efficient. Then, Section 3 defines what we mean with "self-adaptive model checking" before Section 4 discusses the impact of this approach on tools architecture. Section 5 discusses some issues to be solved to enable meta-heuristics to select an appropriate combination of algorithms to solve a verification problem. Section 6 concludes the paper.

2 Information Gathered from the Model Checking Contest

For more than a decade, we observed the emergence of software contests that assess the capabilities of verification tools on complex benchmarks. It is a way to identify the theoretical approaches that are the most fruitful in practice, when applied to realistic examples. Such events motivate the involved community to improve research tools and measure the benefits gained by new improvements.

They are also interesting because they bring representative and shared benchmarks to the involved communities. Moreover, since many tools compete, it is possible to gather and analyze information from the detailed outputs produced by such events. As a typical example, the Model Checking Contest benchmark is growing every year thanks to models proposed by the community. In 2017, it was composed of 78 models from which, thanks to some scaling parameters, 812 instances are derived. Numerous formulas (reachability, CTL, LTL) are also available (a new set is produced every year).

Models proposed by the Model Checking Contest mainly represent concurrent systems and are expressed in PNML [21] (Petri Net markup Language, an ISO/IEC standard to describe Petri net specifications). Some are derived from higher specification languages or translated from code. Some others are natively expressed using Petri nets.

[1] See `http://mcc.lip6.fr`.

Many models come from a wide range of application domains and describe hardware systems, protocols, distributed algorithms, biological systems, etc. Some models are extracted from research papers and denote interesting "theoretical" configurations to be analyzed.

In the Model Checking Contest, tools are confronted to several examinations: StateSpace, UpperBounds, Reachability, CTL and LTL. *StateSpace* requires the tool to compute the full state space of a specification and then provide informations about it. Mandatory information concerns the number of states but tools may also provide additional informations like the number of transitions, the maximum number of tokens per marking in the net and the maximum number of tokens that can be found in a place.

UpperBounds requires the tool to compute as a integer value, the exact upper bound of a list of places designated in a formula (there are 16 formulas per model instance).

Reachability, *CTL*, and *LTL* require the tool to evaluate if formulas are satisfied or not. For each formulas, we consider atomic propositions referring to either the marking of places or the fireability of transitions (16 formulas of each type are provided per model instance). In the reachability examination, there are extra formulas to check if there exist a deadlock.

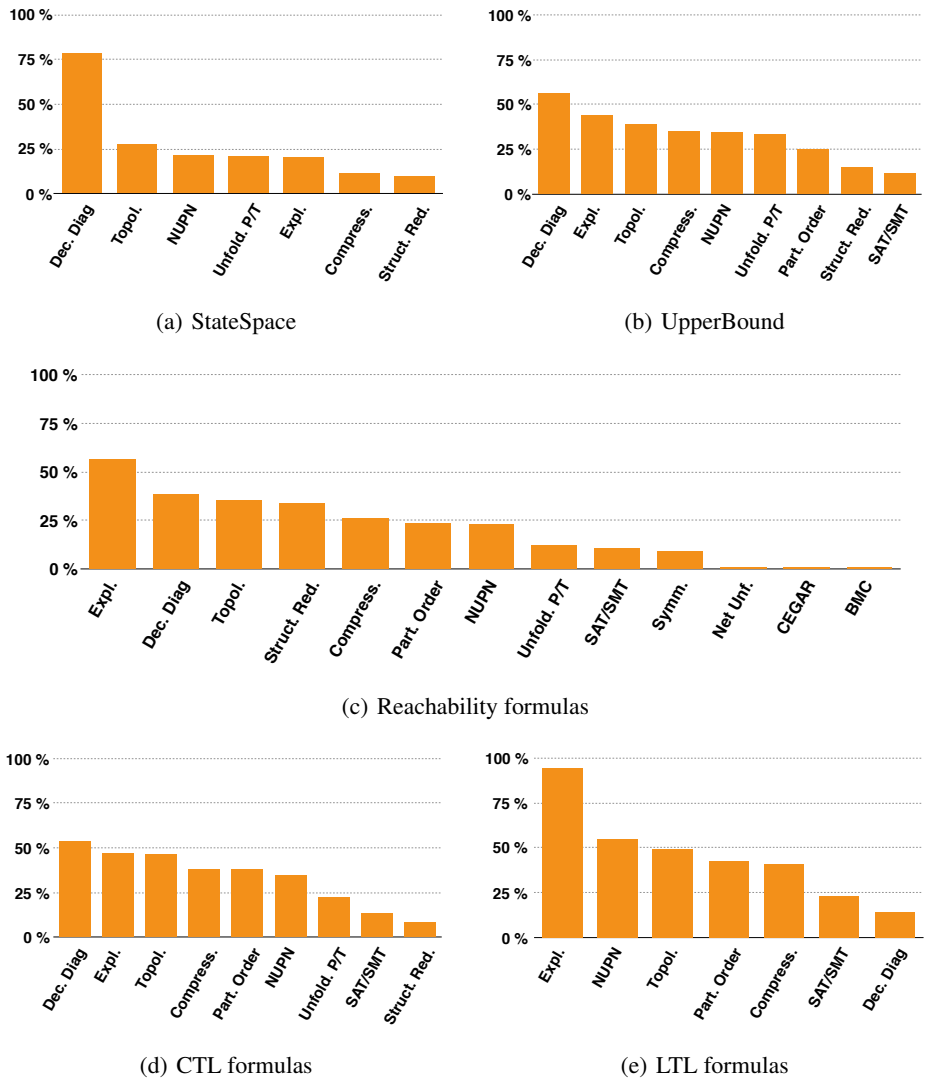
So, the Model Checking Contest could be seen as a way to observe and evaluate the most successful techniques for a given type of model checking activity. To do so, we have analyzed the techniques participating tools reported to use over the three last years of the Model Checking Contest (2015, 2016, and 2017) where data was collected using comparable data formats. All the reported techniques are listed in Table 1.

Figure 1 reports, for the valid answers, the percentage of techniques used by tools to compute examinations (they sometimes use several techniques simultaneously). Unfortunately, these techniques are so far reported per examination, even if, often, an examination contains 16 formulas to be evaluated. However, it is interesting to observe the top winning techniques for each examination categories.

It is easy to see that, based on these raw data, symbolic model checking (based on decision diagrams) is used more often than explicit techniques for the StateSpace,

Technique	**Explanation**
BMC	The tool uses Bounded Model Checking and/or K-induction techniques
CEGAR	The tool uses a CEGAR [9] approach
Compress.	The tool uses some compression technique (other than decision diagrams)
Dec. Diag	The tool uses a kind of decision diagram
Expl.	The tool does explicit model checking
Net Unf.	The tool uses McMillan unfolding [29]
NUPN	The tool exploits the structural information provided in the NUPN [16] format
Part. Order	The tool uses some partial order technique
SAT/SMT	The tool relies on a SAT or SMT solver
Struct. Red.	The tool uses structural reductions (Berthelot, Haddad, etc.)
Symm.	The tool exploits symmetries of the system
Topol.	The tool uses structural informations on the Petri net itself (invariants, etc.)
Unfold. P/T	The tool transforms colored nets into their equivalent P/T

Table 1. List of techniques reported by tools.



(a) StateSpace

(b) UpperBound

(c) Reachability formulas

(d) CTL formulas

(e) LTL formulas

Fig. 1. Cumulated declarations of techniques per successful examination as reported by tools during the Model Checking Contest over the 2015, 2016, and 2017 editions.

the UpperBound, and the CTL examinations. Explicit approaches appear more often in reachability and LTL examinations. Tools also report a large number of additional techniques like compression, partial order, or the use of structural informations to optimize model checking. Some tools also rely on Constraint solving, CEGAR [9] or bounded model checking.

However, this raw data must then be normalized because the number of tools declaring a technique may change from one examination to another as shown in Table 2. For

Technique	StateSpace	UpperBound	Reachability	CTL	LTL
Symbolic tools	22 (71%)	8 (57%)	11 (44%)	10 (63%)	3 (27%)
Explicit tools	9 (29%)	6 (43%)	14 (56%)	6 (27%)	80 (73%)
Total tools	30	13	24	15	10

Table 2. Number of tools relying on Symbolic versus Explicit approaches for participating tools (in the 2015, 2016 , and 2017 editions). Several participations of a given tool are cumulated.

example, the raw value declared for Decision Diagrams in Figure 1(a) must be pondered by the fact that more tools between 2015 and 2017 rely on this technique (so it is naturally reported more often).

Symbolic approaches (based on some type of decision diagram) are usually confronted to explicit ones (often associated with other optimization techniques) in the community since they are in mutual exclusion. It is thus of interest to refine the raw data of Figure 1 by focusing on these two approaches and check which one seems to be the most efficient and in which situations.

Figure 2 summarizes the ponderated ratio between Symbolic and Explicit approaches for the model checking contest examinations proposed in 2015, 2016 an 2017. For each examination, the first large bar shows the ratio between the tools declaring the use of Decision diagrams[2] and those declaring the use of explicit approaches[3]. The two other thinner bars present a "normalized" success rate when considering the respective number of tools using the corresponding technique. All these data are collected for successful examinations only (so the two approaches can claim together 100% of success).

From these normalized data, it appears that symbolic and explicit techniques are quite comparable. For the StateSpace examination, decision diagram technology is a main factor of performance since the full state space must be computed. Figure 1(a) also

[2] Orange or gray in B&W.
[3] Dark blue or black in B&W.

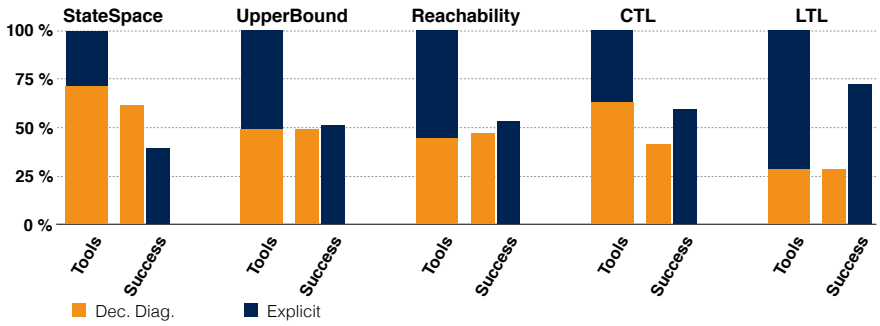


Fig. 2. Normalized measure of the success of Decision Diagram based techniques versus explicit ones (for the 2015, 2016, and 2017 editions)

	Reachability	CTL	LTL	
Satisfied computed formulas	205 478 (51%)	70 824 (45%)	28 773 (23%)	
Unsatisfied computed formulas	197 280 (49%)	84 990 (55%)	97 662 (77%)	**Total**
Total computed formulas	402 758	155 814	126 435	685 007

Table 3. Analysis of the formulas computed by tools over the 2015, 2016 and 2017 editions of the Model Checking Contest. Formulas computed by several tools are cumulated.

shows that topological approaches (for example, based on the exploitation of NUPN[4] data) are quite useful to optimize the encoding of the state space using decision diagrams.

For the UpperBound examination, results are more balanced but slightly in favor of explicit approaches. Similarly to the StateSpace examination, Figure 1(b) outlines an extensive use of topological information (including the use of NUPN data) to compute an appropriate variable order for decision diagrams. Compression mechanisms are also reported to be associated with explicit approaches.

For the evaluation of formulas (reachability, CTL, LTL), there seems to be some advantage to explicit techniques too. This is less clear for reachability formulas, more evident for CTL ones, and particularly true for LTL ones. However, for LTL, we must consider two factors that reduce the relevance of these measures. First, the LTL formula generator used in the Model Checking Contest remains quite basic compared to the one of reachability and CTL formulas (the team is working on this). Second, while reachability and CTL computed formulas are quite balanced between the satisfied and unsatisfied ones (see Table 3), this is not the case for LTL ones (more than $\frac{3}{4}$ are unsatisfied, so that some counter-example might be found rapidly). These two factors probably hinder any conclusion for LTL at this stage.

This simple study, based on he output of a single verification competition, may help to understand how and when one could operate a set of techniques to perform model checking. Unfortunately, even if several involved tools do not come from the Petri net community, the inputs of the Model Checking Contest all represent concurrent systems expressed using Petri nets. So, there might be some bias in the way such models are processed. It would be great if a similar analysis could be done on other verification contests. This is a complex task requiring at least some common glossary and notions to be defined.

3 What is Self-Adaptive Model Checking?

The term "adaptive model checking" [19] was first used to denote a way to learn a model from a component in the context of black-box testing. It was also called later "black box checking" [5].

The context here is totally different. The objective is to integrate some "intelligence" in tools so that they can self-adapt to the *most appropriate combination and configuration* of techniques when verifying a property on a model. In this situation, "appropriate"

[4] NUPN means "Nested-Unit Petri Nets" and is additional information providing some structure to the specification [16]. Some models in the benchmark embed such information.

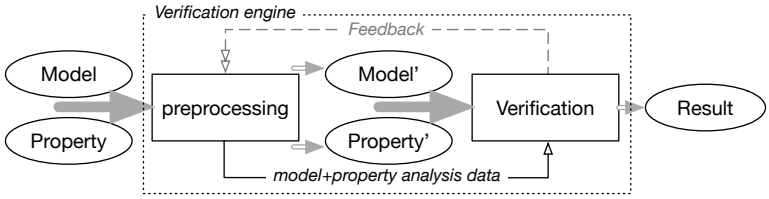


Fig. 3. The self-adaptive model checking process inside a verification engine.

may have several meanings. Among them, let us consider the combinatorial state explosion problem that prevents a non-expert user from verifying a property on a system where some expert would apply a combination of modeling abstractions and the choice of the best model checking engine to solve the problem.

This combinatorial explosion problem is itself difficult to solve because it has to be tackled at different levels. It does not only involve a toolset and must be considered already at the modeling level. This is why it raises methodological issues in the way specifications and properties can be tuned to reduce complexity. Typically, relevant abstractions in the system model with regards to the properties to be verified, may dramatically reduce the verification complexity and should not be ignored.

Self-adaptive model checking must be operated at several stages inside a *verification engine* (seen as a "black box" by users) where various techniques are coordinated to build a verification process.

Figure 3 sketches what could be such a process. First, we consider as input a model and a property (both can be produced automatically or manually). A preprocessing step analyzes the model and the associated property to produce a simplified model and a simplified property, as well as some analysis data. The simplified model must be equivalent to the original one with regards to the property to be checked. Similarly, the simplified property must be equivalent to the original one. Analysis data about the model and its related property can be deduced from structural analysis (*e.g.* hierarchical design of the system, invariants or some properties when the input specification is a Petri nets, or any other information that can be derived from the specification, and possibly the property). In the worst case, the simplified model and/or property are equal to the input model and property. Let us note that, in the Model Checking Contest, such a situation is rare when models are complex (for example, those coming from an industrial case study) and tools implement such type of optimizations.

Once produced, the simplified model and property are then processed by a verification engine which can be adapted using the analysis data.

One can even imagine that a feedback from the way model and property are processed may provide useful information for some later preprocessing. Typically, this could lead to the integration of a CEGAR [9] like loop inside the verification engine itself.

4 Impact on Model Checkers Architecture

This two step process suggests an architecture that is similar to the one of modern compilers (front-end, middle-end, back-end). In fact, it is a trend to adopt this type of architecture in modern model checkers [27].

This trend is illustrated in Figure 4. The idea is to separate the verification engine from the input formalism. Then, the notion of "pivot representation" naturally arises as an intermediate representation between an "upper level", and a a set of "verification engines" able to process this pivot representation. This software architecture brings three major advantages.

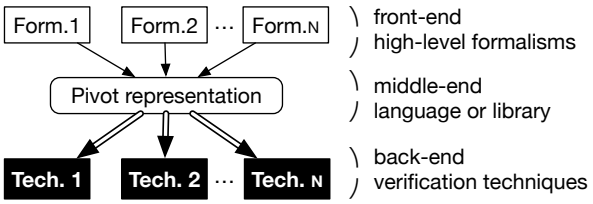


Fig. 4. Software architecture of modern model checkers (from [27]).

First, *it decouples the input (high-level) specification language from its verification*. Then, the specification language designer may work independently from the verification machinery as long as they provide a sound and formal semantics to their notation. This is of particular interest when dealing with numerous input languages, because it does not hinder the access to efficient verification engines thanks to the pivot representation.

Several tools like MC-Kit [32], LTSMin [25], Spot [14], or ITS-Tools [35] already experimented this type of approach. Some of them (like Spot or LTSMin) offer an API to encode the notion of state and the transition relation of a given model. Others (like ITS-Tools) implement an "assembly language" suitable to encode such notions. These solutions showed their efficiency in numerous situations but suffer from an important drawback when it comes to provide feedback to users : they usually lack back-translation mechanisms to show counterexamples in the terms of the input specification. This is one of the main challenge for self-adaptive model checking as soon as one wants to cope with several input formalisms.

The second advantage is that *many preprocessing optimizations can be performed*, either in the front-end (the input language specific ones) or in the middle-end (more generic ones). Optimizations implemented in the middle-end benefit to all the input formalisms supported by the model checker.

We already mentioned tools structured in a way they can easily enable analysis from various input specifications. There is unfortunately no standard pivot representation (neither at an API level nor at a language level) despite the numerous attemps to define interesting languages for verification of industrial-like systems (such as PNML [21],

FIACRE [3], GAL [35], or Promela [22]). Probably, finding a standard suitable to be associated with numerous and different techniques is an important challenge for the community.

The third advantage is that it is then possible to *exploit the technology foreseen to be the most efficient* to process the model and its related property. One can even imagine the back-end to be (automatically) produced on-the-fly from off-the-shelf libraries, assembled to build the most performant model checking engine for a given couple $\langle model, property \rangle$.

It appears from the model checking contest that most winning tools simultaneously activate several techniques to compute properties. Thus, combination of representation techniques (*i.e.* explicit, symbolic, use of different classes of automata), together with reduction algorithms (like partial order, saturation in decision diagrams, etc.) is probably needed in the future. Here, numerous challenges must be addressed to elaborate appropriate and combinable back-ends for verification.

5 The Decision Process

The decision process to select libraries to be assembled to produce an efficient model checking engine is a crucial challenge. A deeper analysis of the involved techniques and their success in identified situations (*e.g.* the use of some operator, some structure of the model or the property, etc.) is required to enable some meta-heuristic that would act as a decision process to perform the assembling of a verification engine.

However, the question of the technique to be used to implement such a meta-heuristic remains. We think that several directions should be investigated:

- An analysis of the input system and property, associated with a dictionary of techniques usable in each situation could be considered. It requires to be aware of the correlations between some characteristics of the couple $\langle model, property \rangle$ and the most efficient technique to solve problems having such characteristics. As an illustration, we can cite the definition of adapted efficient algorithms dedicated to the verification of subcategories of LTL formulas [4]. Unfortunately only a few such situations are identified so far.
- Recent learning techniques have proven their efficiency to take decisions based on the analysis of a large set of data. Unfortunately, we have no evidence that we already have a sufficient amount of unbiased data to operate such a technique.
- The increasing parallelism of modern computers allows us to imagine a portfolio-like implementation of a model checker where several algorithms would concurrently be operated. Unfortunately, such a solution requires massively parallel architectures since the number of possible combinations when mixing algorithms, representations, and implementation matters, grows rapidly.

Of course, even if the complexity of the decision algorithm is more likely to be related to the size of the system instead of the size of its associated state space, it can take a while, thus being of little interest for unsatisfied properties for which a counterexample is found rapidly. However, it is easy to imagine that such an analysis could

be performed in parallel of a first search using some default configuration of the model checking engine. However, for satisfied properties, for the analysis of CTL formulas, or any other situation where the full state space has to be explored to take a decision, the cost of building on-the-fly a dedicated model checking engine would be probably rapidly balanced by the gain on the verification operation itself.

Anyway, at this stage, it is difficult to state which of these approaches will help and produce a self-adaptive model checking tool. We trust there is an interesting problem for the community to deal with.

6 Conclusion

In this paper, we depict *self-adaptive model checking* as a way to increase the efficiency of model checking. It is a mix of methodological, theoretical, and technical visions. Methodological aspects reside in the definition of a typical process including preprocessing (a way to rewrite and simplify the problem) and the use of optimized verification engines, possibly elaborated and compiled on-the-fly for a given couple $\langle model, property \rangle$. Theoretical aspects reside in the fact that theory needs to be extended, for example to enable the combination of several algorithms when it is possible. Technical aspects reside in the definition of some standards (a common pivot representation, a common software architecture, etc.) to enable the sharing of off-the-shelf efficient libraries.

It would also be of interest to share and increase typical benchmarks so that the effect of some algorithm combinations could be explored deeply. Then, more lessons could be gathered from larger experiments on these benchmarks. At this stage, the Model Checking Contest [28] can provide interesting data based on the analysis of the results available for 2015, 2016 and 2017. A first and raw analysis of these data is discussed in the paper. Getting similar information from other similar events should be a goal for the communities involved.

Of course, much work is still needed to complete fully automated self-adaptive model checking. However, the community of model checking already handles many building blocks for this purpose: numerous algorithms, numerous internal representations of the state space (symbolic or explicit, based on a variety of automata), various logics, etc.

So, one can expects that, sooner or later, a new generation of model checking tools will emerge, benefitting from all these expertise, implementation experience, and experimentation.

Self-Adaptive Model Checking is a long-term goal the community should take as an important challenge to deal with.

References

1. Baarir, S., Duret-Lutz, A.: Sat-based minimization of deterministic ω -automata. In: Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference – LPAR. pp. 79–87 (2015)

2. Ben Salem, A., Duret-Lutz, A., Kordon, F., Thierry-Mieg, Y.: Symbolic Model Checking of stutter invariant properties Using Generalized Testing Automata. In: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems – TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 440–454. Springer, Grenoble, France (2014)
3. Berthomieux, B., Bodeveix, J.P., Filali, M., Lang, F., Le Botland, D., Vernadat, F.: The syntax and semantic of fiacre. Tech. Rep. 7264, CNRS-LAAS (2007)
4. Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Computer Aided Verification, 11th International Conference, –CAV. Lecture Notes in Computer Science, vol. 1633, pp. 222–235. Springer (1999)
5. Broy, M., Jonsson, B., Katoen, J., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems, Advanced Lectures, Lecture Notes in Computer Science, vol. 3472. Springer (2005)
6. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Congress on Logic, Method, and Philosophy of Science (1960). pp. 1–12. Stanford University (1962)
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10ˆ20 states and beyond. Inf. Comput. 98(2), 142–170 (1992)
8. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed colored nets and symmetric modeling applications. IEEE Trans. Computers 42(11), 1343–1360 (1993)
9. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B.H., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. 14(4), 583–604 (2003)
10. Clarke, E.M., Jha, S., Marrero, W.R.: Efficient verification of security protocols using partial-order reductions. STTT 4(2), 173–188 (2003)
11. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Towards distributed software model-checking using decision diagrams. In: Computer Aided Verification - 25th International Conference – CAV. pp. 830–845 (2013)
12. Duflot, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A formal analysis of bluetooth device discovery. STTT 8(6), 621–632 (2006)
13. 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: Automated Technology for Verification and Analysis, 9th International Symposium – ATVA. pp. 336–350 (2011)
14. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: International Symposium on Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 9938, pp. 122–129. Springer (2016)
15. Evangelista, S., Haddad, S., Pradat-Peyre, J.: Syntactical colored petri nets reductions. In: Automated Technology for Verification and Analysis, Third International Symposium, – ATVA. pp. 202–216 (2005)
16. Garavel, H.: Nested-unit petri nets: A structural means to increase efficiency and scalability of verification on elementary nets. In: Application and Theory of Petri Nets and Concurrency - 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21-26, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9115, pp. 179–199. Springer (2015)
17. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: 13th International SPIN Workshop – SPIN. Lecture Notes in Computer Science, vol. 3925, pp. 53–70. Springer (2006)
18. Gerth, R.: Model checking if your life depends on it a view from intel's trenches. In: 8th International workshop on Model Checking Software – SPIN. Lecture Notes in Computer Science, vol. 2057, p. 15. Springer (2001)

19. Groce, A., Peled, D.A., Yannakakis, M.: Adaptive model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference – TACAS. Lecture Notes in Computer Science, vol. 2280, pp. 357–370. Springer (2002)
20. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Building efficient model checkers using hierarchical set decision diagrams and automatic saturation. Fundam. Inform. 94(3-4), 413–437 (2009)
21. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. Petri Net Newsletter (originally presented at the 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools – CPN'09) 76, 9–28 (2009)
22. Holzmann, G.: Spin Model Checker, the: Primer and Reference Manual. Addison-Wesley Professional, first edn. (2003)
23. Holzmann, G.J.: Mars code. Commun. ACM 57(2), 64–73 (2014)
24. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., Vergnaud, T.: On the Formal Verification of Middleware Behavioral Properties. In: 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04). pp. 139–157. Elsevier (2004)
25. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: Ltsmin: High-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015)
26. Klai, K., Poitrenaud, D.: MC-SOG: an LTL model checker based on symbolic observation graphs. In: Applications and Theory of Petri Nets, 29th International Conference – PETRI NETS. pp. 288–306 (2008)
27. Kordon, F., Leuschel, M., van de Pol, J., Thierry-Mieg, Y.: Software Architecture of Modern Model Checkers. High Assurance System: Methods, Languages, and Tools (LNCS 10000) p. to appear (2018)
28. Kordon, F., Garavel, H., Hillah, L., Paviot-Adet, E., Jezequel, L., Rodríguez, C., Hulin-Hubard, F.: MCC'2015 - The Fifth Model Checking Contest. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) XI, 262–273 (2016)
29. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Computer Aided Verification, Fourth International Workshop – CAV. Lecture Notes in Computer Science, vol. 663, pp. 164–177. Springer (1992)
30. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transaction of the AMS 141, 1–35 (1969)
31. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Strength-based decomposition of the property büchi automaton for faster model checking. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference – TACAS. pp. 580–593 (2013)
32. Schröter, C., Schwoon, S., Esparza, J.: The model-checking kit. In: Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN. LNCS, vol. 2679, pp. 463–472. Springer (2003)
33. Schwarick, M., Heiner, M.: CSL model checking of biochemical networks with interval decision diagrams. In: Computational Methods in Systems Biology, 7th International Conference, – CMSB. pp. 296–312 (2009)
34. Streett, R.S.: Propositional Dynamic Logic of Looping and Converse is Elementarily Decidable. Information and Control 54(1/2), 121–141 (1982)
35. Thierry-Mieg, Y.: Symbolic model-checking using ITS-Tools. In: TACAS. LNCS, vol. 9035, pp. 231–237. Springer (2015)
36. Wang, F., Schmidt, K., Yu, F., Huang, G., Wang, B.: Bdd-based safety-analysis of concurrent software with pointer data structures using graph automorphism symmetry reduction. IEEE Trans. Software Eng. 30(6), 403–417 (2004)