Developing Model Checkers Using PAT

Yang Liu and Jun Sun and Jin Song Dong

School of Computing
National University of Singapore
{liuyang,sunj,dongjs}@comp.nus.edu.sg

Abstract. During the last two decades, model checking has emerged as an effective system analysis technique complementary to simulation and testing. Many model checking algorithms and state space reduction techniques have been proposed. Although it is desirable to have dedicated model checkers for every language (or application domain), implementing one with effective reduction techniques is rather challenging. In this work, we present a generic and extensible framework PAT, which facilitates users to build customized model checkers. PAT provides a library of state-of-art model checking algorithms as well as support for customizing language syntax, semantics, state space reduction techniques, graphic user interfaces, and even domain specific abstraction techniques. Based on this design, model checkers for concurrent systems, real-time systems, probabilistic systems and Web Services are developed inside the PAT framework, which demonstrates the practicality and scalability of our approach.

1 Introduction

After two decades' development, model checking has emerged as a promising and powerful approach for automatic verification of hardware and software systems. It has been used successfully in practice to verify complex circuit design [3], communication protocols [5] and driver software [2]. Till now, model checking has become a wide area including many different model checking algorithms catering for different properties (e.g., explicitly model checking, symbolic model checking, probabilistic model checking, etc.) and state space reduction techniques (e.g., partial order reduction, binary decision diagrams, abstraction, symmetry reduction, etc.).

Unfortunately, several reasons prevent many domain experts, who may not be experts in the area of model checking, from successfully applying model checking to their application domains. Firstly, it is nontrivial for a domain expert to learn a general purpose model checker (e.g., NuSMV [3], SPIN [5] and so on). Secondly, general purpose model checkers may be inefficient (or insufficient) to model domain specific applications, due to lack of language features, semantic models or data structures. For example, multi-party barrier synchronization or broadcasting is difficult to achieve in the SPIN model checker. Lastly, the level of knowledge and effort required to create a model checker for a specific domain is even higher than applying existing ones.

To meet the challenges of applying model checking in new application domains, we propose a generic and extensible framework called PAT (Process Analysis Toolkit) [1], which facilitates effective incorporation of domain knowledge with formal verification

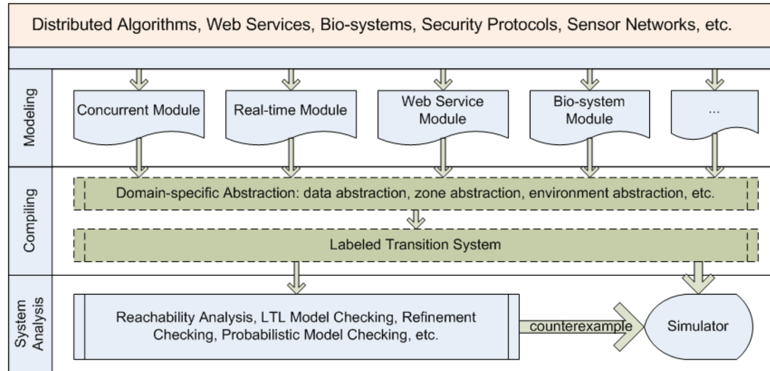

Fig. 1. PAT Architecture

using model checking techniques. PAT is a self-contained environment to support composing, simulating and reasoning of system models. It comes with user friendly interfaces, a featured model editor and an animated simulator. Most importantly, PAT implements a library of model checking techniques catering for checking deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions [13], refinement checking [11] and probabilistic model checking. Advanced optimization techniques are implemented in PAT, e.g., partial order reduction, process counter abstraction [16], bounded model checking [14], parallel model checking [8] and probabilistic model checking. PAT supports both explicit state model checking and symbolic model checking (based on BDD or SAT solver). We have used PAT to model and verify a variety of systems [6]. Previously unknown bugs have been discovered [7]. The experiment results show that PAT is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in some cases.

2 Architecture Overview

PAT was initially designed to support a unified way of model checking under fairness [13]. Since then, PAT has been extended significantly and completely re-designed. We have adopted a layered design to support analysis of different systems/languages, which can be implemented as plug-in modules. Fig. 1 shows the architecture design of PAT. For each supported domain (e.g., distributed system, real-time system, service oriented computing and so on), a dedicated module is created in PAT, which identifies the (specialized) language syntax, well-formness rules as well as formal operational semantics. For instance, the CSP module is developed for the analysis of concurrent system modeled in CSP# [12]. The operational semantics of the target language translates the behavior of a model into LTS (Labeled Transition Systems)[1] at runtime. LTS serves as an implicitly shared internal representation of the input models, which can be automatically explored by the verification algorithms or used for simulation. To perform model checking on LTSs, the number of states in the LTSs needs to be finite. For systems with infinite behavior (e.g., real time clocks or infinite number of processes),

[1] To be precise, it is a Markov Decision Process when probabilistic choices are involved.

abstraction techniques are needed. Examples of abstraction techniques include data abstraction, process counter abstraction, clock zone abstraction, environment abstraction, etc. The verification algorithms perform on-the-fly exploration of the LTSs. If a counterexample is identified during the exploration, then it can be animated in the simulator. This design allows new modules to easily be plugged in and out, without recompiling the core system, and the developed model checking algorithms (mentioned in Section 1) to be shared by all modules. This design achieves *extensible architecture* as well as *module encapsulation*. We have successfully applied this framework in development of four different modules, each of which targets a different domain. 1). The concurrent system module is designed for analyzing general concurrent systems using a rich modeling language CSP# [12], which combines high-level modeling operators with programmer-favored low-level features. 2). The real-time system module supports analysis of real-time systems with compositional behavioral patterns (e.g. $timeout$, $deadline$) [15]. Instead of explicitly manipulating clock variables, time related constructs are designed to build on implicit clocks and discretized using clock zone abstraction [15]. 3). The Web Services (WS) module offers practical solutions to the conformance checking and prototype synthesis between WS Choreography and WS Orchestration. 4). The probabilistic module supports the modeling and verification of systems exhibiting random or probabilistic behavior.

3 Manufacturing Model Checkers

In this section, we discuss how to create a customized model checker for a new domain using the PAT framework with the help of its predefined APIs, examples and software packages. A domain often has its specific model description language. It is desirable that the domain experts can input their models using their own languages. There are three different ways of supporting a new language in PAT.

- The easiest way is to create a syntax rewriter from the domain specific language to an existing language. This is only recommended if the domain language is less expressiveness than the existing languages. For example, we have developed translators from Promela/UML state diagram to CSP#. Comparing with other tools, programming a translator is straightforward in PAT. Because PAT has open APIs for its language constructs, users only need to generate the language constructs objects using these APIs, which can guarantee that the generated syntax is correct. This approach is simple and requires little interaction with PAT codes. However, translation may not be optimal if special domain specific language features are present. Furthermore, reflecting analysis results back to the domain model is often non-trivial.
- The second way is to extend an existing module if the input languages are similar and yet with a few specialized features. For example, the probabilistic module is designed to extend the concurrent system module with one additional language feature, i.e., probabilistic choices. Knowledge about existing modules is required and a new parser may be created for the extended language features.
- The third way is to create a new module in PAT. In this case, users firstly need to develop a parser according to the syntax. The parser should generate a model con-

sisting of ASTs of language construct classes, which encode their operational semantics. Abstract classes[2] for system states, language construct classes and system model are pre-defined in PAT with (abstract) signature methods for communications with verification algorithms and user interface interactions. Users only need to develop concrete classes in the new module by inheriting the abstract classes. This approach is the most complicated compared with the first two. Nevertheless, this approach gives the most flexibility and efficiency. It is difficult to quantify the effort required to build a high-quality module in PAT. Experiences suggest that a new module can be developed in months or even weeks in our team. This approach is feasible for domain experts who have only the basic knowledge on model checking. This is because model checking algorithms and state space reduction techniques are separated from the syntax and semantics of the modeling language.

It is possible that a domain may have its own specialized properties to verify and specified model checking algorithms. Our design allows seamless integration of new model checking algorithm and optimization techniques by inheriting base assertion class and implementing its API. Furthermore, supporting functions, like LTL to Büchi, Rabin, Streett automata conversion, are provided in PAT to ease the development of new algorithms. For instance, we have successfully developed the algorithms for divergence checking, timed refinement checking in real-time system module and new deadlock and probabilistic reachability checking. Furthermore, PAT facilitates customized state encoding by defining the interfaces methods in system state class. Different verification algorithms using different state encoding are developed. Currently, PAT supports explicitly state encoding using hash table and symbolic state representation using BDD. The choice of the encoding is made by the users in the user interface at runtime.

4 Performance Evaluation

PAT is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in some cases. Experimental results for LTL verification under fairness and refinement checking are presented in Fig. 2 as an indication of our effort on optimizing the model checking algorithms.

The table on the left shows the verification results on recently developed leader election protocols with different topologies, where the correctness (modeled using LTL formula) of these protocols requires different notions of fairness. Firstly, PAT usually finds counterexamples quickly. Secondly, verification under event-level strong fairness (ESF) is more expensive than verification with no fair, event-level weak fairness (EWF) or strong global fairness (SGF). Lastly, PAT outperforms SPIN for the fairness verifications. SPIN increases the verification time under weak fairness by a factor that is linear in the number of processes. SPIN has no support for strong fairness or SGF. PAT offers comparably better performance on verification under weak fairness and makes it feasible to verify under strong fairness or SGF.

In addition to temporal logic verification, PAT offers capability of refinement checking (i.e. language inclusion checking). The table on the right shows the performance us-

[2] Detailed explanation and usages of the abstract classes are available in PAT's user manual.

Model	Size	EWF			ESF		SGF	
		Res.	PAT	SPIN	Res.	PAT	Res.	PAT
LE_C	5	Yes	4.7	35.7	Yes	4.7	Yes	4.1
LE_C	6	Yes	26.7	229	Yes	26.7	Yes	23.5
LE_C	7	Yes	152	1190	Yes	152	Yes	137
LE_C	8	Yes	726	5720	Yes	739	Yes	673
LE_T	7	Yes	1.4	7.6	Yes	1.4	Yes	1.4
LE_T	9	Yes	10.2	62.3	Yes	10.2	Yes	9.6
LE_T	11	Yes	68.1	440	Yes	68.7	Yes	65.1
LE_T	13	Yes	548	3200	Yes	573	Yes	529
LE_OR	3	No	0.2	0.3	No	0.2	Yes	11.8
LE_OR	5	No	1.3	8.7	No	1.8	-	-
LE_OR	7	No	15.9	95	No	21.3	-	-
LE_R	4	No	0.3	<0.1	No	0.7	Yes	19.5
LE_R	5	No	0.8	<0.1	No	2.7	Yes	299
LE_R	6	No	1.8	0.2	No	4.6	-	-
LE_R	7	No	4.7	0.6	No	9.6	-	-
LE_R	8	No	11.7	1.7	No	28.3	-	-
TC_R	3	Yes	<0.1	<0.1	Yes	<0.1	Yes	<0.1
TC_R	5	No	<0.1	<0.1	No	<0.1	Yes	0.6
TC_R	7	No	0.2	0.1	No	0.2	Yes	13.7
TC_R	9	No	0.4	0.2	No	0.4	Yes	640

Models	N	Property	Result	PAT	FDR
Dining Philosophers	6	P refines S	true	0.86	0.07
Dining Philosophers	8	P refines S	true	13.7	0.07
Dining Philosophers	10	P refines S	true	430	0.11
Reader/Writers	12	P refines S	true	< 1	0.81
Reader/Writers	14	P refines S	true	< 1	6.91
Reader/Writers	16	P refines S	true	< 1	81.2
Reader/Writers	200	P refines S	true	77.5	-
Milner's Cyclic Scheduler	11	P refines S	true	< 1	89.4
Milner's Cyclic Scheduler	12	P refines S	true	< 1	419
Milner's Cyclic Scheduler	13	P refines S	true	< 1	-
Milner's Cyclic Scheduler	200	P [T= S	true	60.4	-
5-valued *register*	2	P refines S	true	44.9	NA
6-valued *register*	2	P refines S	true	297	NA
stack of size 14	2	P refines S	true	99.4	NA
stack of size 2	3	P refines S	true	4321	NA
buggy queue of size 10	2	P refines S	false	6.87	NA
buggy queue of size 20	2	P refines S	false	41.1	NA
mailbox of 3 operations	2	P refines S	true	27.8	NA
mailbox of 4 operations	2	P refines S	true	954	NA
SNZI of size 2	2	P refines S	true	322	NA
SNZI of size 3	3	P refines S	true	6214	NA

Fig. 2. Experiment results on LTL verification under fairness assumption and refinement checking

ing benchmark systems as well as newly developed concurrent algorithms. In the classic readers/writers problem, reduction in PAT is very effective so that PAT can handle a few hundreds readers/writers. In the Milner's cyclic scheduling algorithm, multiple processes are scheduled in a cyclic fashion. PAT is effective for this model to handle hundreds of processes. For models with complicated data variables (like scalable non-zero indicator SNZI, see [6] for the details of the examples), PAT is able to show the linearizability of these examples using refinement checking [6]. FDR [10] performs extremely well for Dining Philosophers because of the compression strategy developed for some specialized models. For other examples, PAT is much faster than FDR. Limited by the modeling language, FDR is rather difficult to model distributed systems like Stack, mailbox and SNZI. In addition, PAT supports timed refinement checking, which is beyond existing refinement checkers. In summary, PAT offers a set of well-optimized model checking languages as well as a framework for developing new model checkers.

5 Discussion and Summary

As a temporal logic model checker, PAT is related to the tools like NuSMV [3] and SPIN [5]. Compared to these tools, PAT serves a generic framework for manufacturing model checkers. It complements existing model checkers with specialized algorithms for (timed/untimed) refinement checking [11], verification under fairness constraints (with counter abstraction [16]), etc. PAT has a comparative performance with existing state-of-art tools, and even out-performs them on some cases. Bogor [4] and LTSA [9] are two extensible model checker developed as a plug-in of Eclipse. Bogor allows users to extend the base language to support new language features, but cannot be fully customized with desired syntax and semantic models. LTSA compiles the input language FSP (based on Process Algebra) into LTS, which is similar to PAT. However all the modules in LTSA adopt the translation approach to convert the input model (e.g., Message Sequence Chart and Web Service) into FSP models. Compared with these two, our

approach takes one step further to allow the development of fully customized model checkers. Furthermore, the supported libraries in PAT offer user advanced model checking techniques like real-time verification and probabilistic model checking, which are absent in Bogor and LTSA.

Compared to [13], we redesigned the system to separate the GUI, verification algorithms and modeling languages. Each modeling language is encapsulated into a standalone package, which makes the system extensible. Furthermore, we have added the support for real-time and probabilistic systems. The enhancement is dramatic. Starting from 2007, PAT has come to a stable stage with solid testing and various applications. More than 60 built-in examples and hundreds of test cases are embedded in PAT. PAT has been used by a number of institutions as a research or educational tool. The main objective of PAT is to bring sophisticated model checking techniques to a variety of domains. The existing modules and on-going modules under development have shown the usefulness and feasibility of this framework.

References

1. PAT: Process Analysis Toolkit. http://pat.comp.nus.edu.sg/.
2. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM 2004*, pages 1–20. Springer, 2004.
3. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV 2002*, pages 359–364, 2002.
4. M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In *CAV 2005*, pages 148–152, 2005.
5. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Wiley, 2003.
6. Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Lineariability via Refinement. In *FM 2009*, pages 321–337, 2009.
7. Y. Liu, J. Pang, J. Sun, and J. Zhao. Efficient Verification of Population Ring Protocols in PAT. In *TASE 2009*, pages 81–89, 2009.
8. Y. Liu, J. Sun, and J. S. Dong. Scalable Multi-Core Model Checking Fairness Enhanced Systems. In *ICFEM 2009*, pages 426–445, Dec 2009.
9. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.
10. A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.
11. J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA 2008*, pages 307–322. Springer, 2008.
12. J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE 2009*, pages 127–135. IEEE Computer Society, 2009.
13. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009.
14. J. Sun, Y. Liu, J. S. Dong, and J. Sun. Bounded Model Checking of Compositional Processes. In *TASE 2008*, pages 23–30. IEEE Computer Society, 2008.
15. J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction. In *ICFEM 2009*, pages 581–600, Dec 2009.
16. J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair Model Checking of Parameterized Systems. In *FM 2009*, pages 123–139, 2009.