Build Your Own Model Checker in One Month

Jin Song Dong
National University of Singapore
dongjs@comp.nus.edu.sg

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

Yang Liu
Nanyang Technological University
yangliu@ntu.edu.sg

***Abstract*—Model checking has established as an effective method for automatic system analysis and verification. It is making its way into many domains and methodologies. Applying model checking techniques to a new domain (which probably has its own dedicated modeling language) is, however, far from trivial. Translation-based approach works by translating domain specific languages into input languages of a model checker. Because the model checker is not designed for the domain (or equivalently, the language), translation-based approach is often ad hoc. Ideally, it is desirable to have an optimized model checker for each application domain. Implementing one with reasonable efficiency, however, requires years of dedicated efforts.**

In this tutorial, we will briefly survey a variety of model checking techniques. Then we will show how to develop a model checker for a language combining real-time and probabilistic features using the PAT (Process Analysis Toolkit) step-by-step, and show that it could take as short as a few weeks to develop your own model checker with reasonable efficiency. The PAT system is designed to facilitate development of customized model checkers. It has an extensible and modularized architecture to support new languages (and their operational semantics), new state reduction or abstraction techniques, new model checking algorithms, etc. Since its introduction 5 years ago, PAT has attracted more than 2500 registered users (from 500+ organisations in 60 countries) and has been applied to develop model checkers for 20 different languages.

I. Introduction

Software development has entered a mass production era. To ensure quality, software verification is becoming a compulsory step in the software development life cycle, especially for safety and critical systems. Among the principal validation/verification methods (e.g., simulation, testing and theorem proving), model checking [3] has emerged as a promising and powerful approach to automatically verify software systems, e.g., complex circuit design, communication protocols, driver software, software process models, software requirement models, architectural frameworks, product lines and system implementations.

Model checking is the application of an automatic process to verify whether a *model* satisfies a *property* by exhaustively exploring the *state space* of the model. Till now, it has become a wide area including many different approaches (e.g., explicit model checking and symbolic model checking) catering for different properties (e.g., temporal logics, refinement relationship, etc.) and state space reduction techniques (e.g., partial order reduction, symmetry reduction, etc.). Applying model checking in a new domain requires in-depth understanding of model checking techniques. Unfortunately, the complexity prevents many domain experts, who may not be experts in the area of model checking, from successfully applying model checking to their domains.

In this tutorial, we cover the basic knowledge about model checking and explain how to adopting model checking in new application domains. This tutorial consists of two parts. The first part briefly surveys the state-of-the-art model checkers and discusses the challenge in applying model checking techniques. The second (and the main) part details what are necessary steps to build a model checker of your language. In particular, we will show how the PAT framework is designed to help using a concrete example: how to step-by-step develop a model checker for hierarchical real-time probabilistic systems.

II. Model Checking Systems Overview

Many model checkers have been developed and successfully applied to practical systems, among which the most noticeable ones include SPIN, NuSMV, FDR, Uppaal, PRISM and Java Pathfinder. With the successful story of the SLAM project and the Intel i7 chip verification, model checking keeps marching its way into new domains/applications. This tutorial will cover the basic concepts of model checking, including Kripke structures, safety and liveness properties, temporal logics, basic model checking algorithms and reduction techniques (partial order reduction and symmetry reduction). In the end, a detailed comparison on the state-of-the-art model checkers will be given.

There are several approaches in applying model checking to a new domain (and often a new language). A common approach is based on translating domain specific languages into input languages of existing model checkers. The advantage of this approach is that the domain experts need little knowledge on model checking. This approach is, however, often less than ideal. Firstly, though model checking is known to be a "push-button" technique, applying a model checker in its most effective setting can be very tricky. A precise understanding of the various verification options is often necessary, which can be very involved. Secondly, existing model checkers may be inefficient or insufficient to model domain specific applications, often due to lack of language features. A possible remedy is to extend an existing model checker so as to support new language features, new reduction techniques, or even new model checking algorithms. It is, however, often more challenging than expected. Model checkers are complicated and highly coupled (for the sake of efficiency). Understanding the (complete) source code of a model checker is extremely difficult, and often impossible without proper documentation.

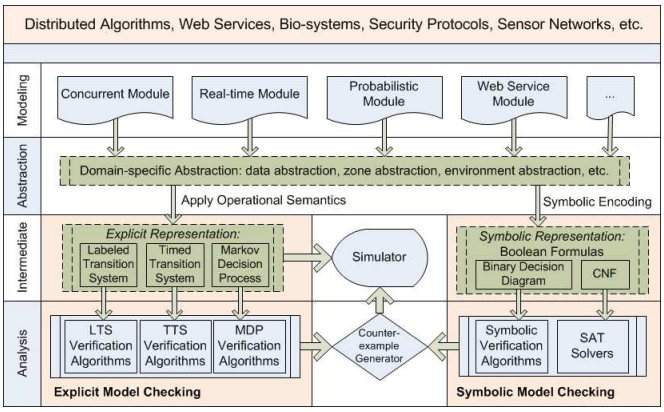

Fig. 1. PAT Architecture

Another approach is to develop a model checker from scratch. In theory, this is ideal as the dedicated model checker can support the domain specific language, and possible domain specific state-reduction/abstraction techniques as well as model checking algorithms. It is, however, probably the most challenging. The basic functionalities of a model checker include language parsing, simulation, verification, state reduction, counterexample generation and display, etc. To finish all with reasonable soundness guarantee often takes years of effort. Notice that most of the established model checkers (e.g., SPIN, NuSMV, UPPAAL and PRISM) are the result of decades' development.

III. DEVELOP DOMAIN SPECIFIC MODEL CHECKER USING PAT

The PAT (short for Process Analysis Toolkit) framework [6], [16], [8], [9] is designed to solve the problems encountered in applying model checking techniques in new domains or languages. It is a self-contained environment for system modeling, simulation and verification. It comes with cross-platform support, internationalization with 6 languages, user friendly graphical interfaces, featured model editors and an animated simulator. Most importantly, it adopts a layered architecture so as to achieve extensibility.

Figure 1 above shows the layered architecture of PAT [9]. Each layer is explained as follows.

- **Modeling Layer** The top level of the architecture is the supported application domains (e.g., distributed system, service oriented computing, security protocols and so on). For each application domain, modeling layer identifies the domain specific language syntax, well-formedness rules as well as formal operational semantics, which are all encapsulated in a separate *module*. The main components in modeling layer are the language parser and model components (including syntax classes, variables, channels, etc.). The (operational) semantics of the language shall be implemented in the syntax classes of the language constructs, which can be either explicit state representation or symbolic representation using Boolean formulae.
- **Abstraction Layer** Model checking techniques generally only work with finite state systems. When a system has infinitely many states according to its *concrete* operational semantics, (automated) state abstraction is essential. For example, a real-time system always has infinitely many states since there is an arbitrary small interval between any two time points. Hence abstraction techniques like zone abstraction [4] can be used to generate finite systems. Even when the state space is finite, effective abstraction/reduction techniques may reduce the state space significantly. For example, process counter abstraction [19] can group identical processes using process counter variables and ignore process identifiers (if they are irrelevant). In PAT, the abstraction layer implements abstraction techniques as independent functions, which map a concrete state to an abstract state. These functions are invoked during the state space exploration to generate abstract states. Abstraction/reduction techniques, like partial order reduction, are language or algorithm dependent, which are then treated differently.
- **Intermediate Representation Layer (IRL)** IRL contains different semantic models supported in PAT. Each semantic model defines a state interface class with methods to drive the state space exploration. After compilation, the input model is converted to an initial state interface class. Then the state space can be generated on-the-fly starting from the initial state by following the operational semantics (and applying abstraction/reduction techniques). PAT supports three semantic models, i.e., Labeled Transition System, Timed Transition System and Markov Decision Processes.
  For explicit model checking, the state interface class has a number of operations, which allow the underlying model checking algorithms to drive the execution of the system and collect information from system states. The state interface can also be used by the simulator to show the system state space graphically.
  For symbolic model checking, different operations are defined so that a symbolic representation is generated to capture the language semantics, usually in the form of Boolean formulae. The Boolean formulae are usually stored in the form of Binary Decision Diagram (BDD) for symbolic model checking or Conjunctive Normal Form (CNF) for bounded model checking.
- **Analysis Layer** This layer mainly contains reusable model checking algorithms. In the explicit model checking approach, a set of verification algorithms have been developed for each semantic model in IRL. For example, deadlock checking, reachability checking, LTL verification with fairness assumptions [18], [7], refinement checking [13], [21] have been developed for LTS. The verification algorithms only invoke state interface to explore the state space. Therefore, the modeling language is separated from the verification algorithms completely. If the verification result is false, a counterexample is

produced, which can be visualized via the simulator. For the symbolic model checking approach, symbolic verification algorithms [12], [11], [10] are developed for the generated BDD encoding of the system. Alternatively, SAT solvers can be used for solving CNF equations for bounded model checking [17].

PAT has been applied to model and verify a variety of systems, ranging from recently proposed concurrent algorithms [5], sensor networks [22], security protocols [1] to real-world systems like the multi-lift and pacemaker systems. Previously unknown bugs have been discovered [15], [1]. Experiment results (can be found in PAT website) 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. PAT has attracted more than 2500 registered users. Mostly important, PAT's extensibility has been evidenced by the number of different languages that it supports: 20, including CSP# [14], NesC [22], PRTS [20], Stateflow [2], Timed Automata, etc. In the following, we illustrate what are the necessary steps to build a customized model checker for your own.

1) **Define Syntax of Your Language** The first step is to define the syntax of a modeling language for your target systems. In this tutorial, the targeted modeling language, PRTS, is a hierarchial language combining concurrency, real-time and probabilistic features. The detailed syntax can be found in PRTS module in PAT user manual. With the syntax constructs, the *Module Generator* tool in PAT can generate the code skeleton for the module including the necessary interfaces and languages syntax classes.
2) **Encode Your Language Semantics** The second step is to define the operational semantics of your language. This is achieved again through implementing pre-defined state interface. The interface is called $Configuration$, which defines the notion of global states. It should compactly encapsulate every relevant *varying* elements in the systems. In $Configuration$ class. the method $Next$ returns a set of $next$-configurations given a current configuration of the system. Depending the semantic model, the next configurations may be in different forms. For instance, if the semantic model is labeled transition transition, then it is a set of configurations. For the targeted PRTS language in this case study, the semantic model is Markov Decision Processes, then it is a set of distributions. Furthermore, the $Configuration$ class is often the place where domain specific state reduction/abstraction techniques can be implemented.
3) **Extend PAT's Model Checking Libraries** It is possible that a domain may have specialized properties, which require dedicated model checking algorithms; or the domain has certain property which implies a more efficient algorithm for checking certain property. PAT's design allows seamless integration of new model checking algorithms and optimization techniques. To create a new property, users need to create a new assertion class, inheriting the base $Assertion$ class and implementing its pre-defined interface. In this example, we will implement an SCC-based algorithm in PAT to verify LTL property with fairness assumptions and show that it outperforms SPIN significantly [16].

IV. Value and Scope

The audiences will learn every step of developing a model checker. The intended audiences of this tutorial are academics, graduate students, researchers, software architects and analysts. We particularly welcome those who want to apply formal methods (especially model checking techniques) into a their own application domains.

References

[1] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS*, February 2013.

[2] C. Chen, J. Sun, Y. Liu, J. Dong, and M. Zheng. Formal modeling and validation of stateflow diagrams. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:653–671, 2012.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[4] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[5] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Linearizability via Refinement. In *FM*, pages 321–337, 2009.

[6] Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *ICSE Companion*, pages 919–920. ACM, 2008.

[7] Y. Liu, J. Sun, and J. S. Dong. Scalable Multi-Core Model Checking Fairness Enhanced Systems. In *ICFEM*, pages 426–445, 2009.

[8] Y. Liu, J. Sun, and J. S. Dong. Developing model checkers using pat. In *ATVA*, pages 371–377, 2010.

[9] Y. Liu, J. Sun, and J. S. Dong. PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *ISSRE*, pages 190–199, 2011.

[10] T. K. Nguyen, J. Sun, Y. Liu, and J. S. Dong. A model checking framework for hierarchical systems. In *ASE*, pages 633–636, 2011.

[11] T. K. Nguyen, J. Sun, Y. Liu, and J. S. Dong. Symbolic model-checking of stateful timed csp using bdd and digitization. In *ICFEM*, pages 398–413, 2012.

[12] T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu. Improved bdd-based discrete analysis of timed systems. In *FM*, pages 326–340, 2012.

[13] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA*, pages 307–322, 2008.

[14] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE'09*, pages 127–135. IEEE Computer Society, 2009.

[15] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and E. Andre. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *TOSEM*, 22(1), 2012.

[16] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 702–708, 2009.

[17] J. Sun, Y. Liu, J. S. Dong, and J. Sun. Bounded Model Checking of Compositional Processes. In *TASE*, pages 23–30, 2008.

[18] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *ICFEM*, pages 318–337, 2008.

[19] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM*, pages 123–139, 2009.

[20] J. Sun, Y. Liu, S. Song, J. S. Dong, and X. Li. Prts: An approach for model checking probabilistic real-time hierarchical systems. In *ICFEM*, pages 147–162, 2011.

[21] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In *ICFEM*, pages 364–380, 2012.

[22] M. Zheng, J. Sun, D. Sanán, Y. Liu, J. S. Dong, and Y. Gu. Towards bug-free implementation for wireless sensor networks. In *SenSys*, pages 407–408, 2011.