Model Checking in the Absence of Code, Model and Properties
(A Preliminary Study)

David Lo
National University of Singapore
dlo@comp.nus.edu.sg

Siau-Cheng Khoo
National University of Singapore
khoosc@comp.nus.edu.sg

Abstract

Model checking is a major approach in ensuring software correctness. It verifies a model converted from code against some formal properties. However, difficulties and programmers' reluctance to formalize formal properties have been some hurdles to its widespread industrial adoption. Also, with the advent of commercial off-the-shelf (COTS) components provided by third party vendors, model checking is further challenged as often only a binary version of the code is provided by vendors. Interestingly, latest instrumentation tools like PIN and Valgrind have enable execution traces to be collected dynamically from a running program. In this preliminary study, we investigate what can be done with model checking tools when code, model and properties are not available and the only available input is execution traces. Specifically, we combine studies on learning automata from traces and learning temporal properties from traces. The preliminary study suggests an automatic way to discover bugs using model checking tools when only execution traces are available.

1. Motivation and Basic Idea

Model checking verifies a form of automata (Buchi [2] or Probabilistic [8]) against some temporal logic properties. The model is usually extracted from code [3] or drafted manually while the properties to be verified are usually given by the user. Besides ensuring absolute correctness, model checking has also been used in finding bugs [7]. The latter is the focus of our study.

The difficulties and programmers' reluctance in formulating a set of formal properties have been some of the barriers to the widespread adoption of model checking [1, 9]. Adding the fact that software changes throughout its lifespan [13], the model checking process is further challenged. As a system changes and features are added or removed, there is a constant need to add, modify or remove properties to ensure the effectiveness of model checking in finding bugs and ensuring software correctness. Hence, a tool to automatically extract or infer program properties as a program changes over time is desirable [1].

Commercial Of-The-Shelf (COTS) components are components purchased from third party vendors to reduce software development time and effort [21]. Due to the market demand to develop large software solutions fast, COTS components have become commonly used in the industry. With the advent of COTS components provided by third party vendors, model checking is further challenged as often only a binary version of the code is provided by vendors. Interestingly, latest instrumentation tools like PIN [20] and Valgrind [23] have enabled execution traces to be collected from running program dynamically without the need of a source code. It is certainly desirable to have a tool that can automatically learn a model either from a binary code or traces for model checking purpose.

Recently, there have been active interests in mining program specifications from traces (*e.g.,* [1, 18, 24, 16, 19]). Two types of formal specifications are often mined: automata [1, 18] and Linear Temporal Logic (LTL) [10] rules or properties [24, 16, 19]. Let us refer to the two approaches as *automata-based and rule-based specification mining* respectively[1]

In [1], Ammons *et al.* learn an automata expressing a specification of X11 Windowing Toolkit library from traces. We have extended the above work in [18] in order to improve the accuracy, scalability and robustness of the mining process. In [24], Yang *et al.* learn a family of two-event LTL rules (*e.g.,* $\langle lock \rangle \rightarrow \langle unlock \rangle$) from traces. We have extended the above work in [16, 19] to mine a family of LTL rules of arbitrary lengths from traces. Each of the mined rules has the following format: "Whenever a series of events pre occurs, eventually another series of events $post$ will occur". The rules correspond to two families of frequently used LTL properties for model checking [4].

Comparing studies on automata-based and rule-based specification mining, we observe the following:

> Automata-based specification mining learns a *global model* of the input traces while rule-based specification mining extract *strong properties* observed in the traces.

From the above observation, we propose a novel approach to combine the benefits of learning automata (*i.e.*, the model) and rules (*i.e.*, the properties) from traces and integrate them into a model checking framework. Our purpose is bug finding and is based on the assumption made by Engler *et al.* [5], namely, bugs usually correspond to deviant behaviors. Simply put, if a program behaves in one way 99% of the time and the opposite 1% of the time, the latter is potentially a bug.

The outline of the paper is as follows: Section 2 presents our novel end-to-end mining-model-checking framework. Section 3 discusses related issues on learning models from traces. We finally conclude and present some future work in Section 4.

2. End-to-end Framework

Our end-to-end mining-model checking framework is shown in Figure 1.

We start with a set of test cases and a program. The program can then be instrumented by various approaches. Java class files can be instrumented using Java Runtime Analysis Toolkit [11] (*c.f.*, [24, 18]). Binary C files can be instrumented using Executable Editing Library [12] (*c.f.*, [1]) or Kvasir [22] (*c.f,* [6]). Latest instrumentation tools PIN [20] and Valgrind [23] has enable execution traces to be collected dynamically from a running program.

Running an instrumented code over the test cases will generate a set of execution traces. A trace can be considered as a sequence of events, each corresponding to: a statement that is executed or a method that is invoked, etc. Traces can then be abstracted to a level of interest. Abstraction takes a trace and converts it to a sequence of symbols from an alphabet. Two or more similar events can be represented as or unified to the same symbol. Some events can also be ignored as non-interesting ones.

For example, one might be interested in the orderings of method calls and would like to verify these. Non-method calls recorded

[1] The terms come from two previous studies in [17, 1].

during the trace generation process can then be removed. One can also choose to ignore non-interesting behaviors (*e.g.*, calls to a method writing to a log file). Furthermore, one can unify method invocations of object instances of classes belonging to the same inheritance hierarchy to the same symbol. One can also abstract away, group together or consider in detail parameters passed to a method when it was called.

After the abstraction process, we obtain abstract traces and it corresponds to a multi-set of sequences of symbols. These abstracted traces can then be fed to the model generation and property extraction processes. The result is a model in the form of an automata and a set of strong LTL properties observed in the traces ready as inputs for model checking. A strong property is one that is observed frequently and with a high confidence (see [16, 19]). The set of LTL rules minable by our technique proposed in [16, 19] can be represented in the Backus-Naur Form (BNF) as follows:

$$\begin{array}{rl} rules := & G(prepost) \\ prepost := & event \rightarrow post | event \rightarrow XG(prepost) \\ post := & XF(event) | XF(event \wedge XF(post)) \end{array}$$

Model checking will verify the generated model against the LTL rules. It will either confirm its verification or return a counter example identifying a potential bug.

3. Discussion

Aside from addressing situations where the source code is unavailable (as is the case with COTS components), generating model from traces also has its own forte.

Models extracted from traces are especially suitable for capturing behaviors of user-input-dependent programs. Also, the model can focus on a behavior of interest, rather than on all possible behaviors a code might exhibit. The model will also tend to be smaller and faster to check. Many programs are not built from scratch, rather often they are adapted (via copy and paste) from existing programs [14]. After several chains of adaptations, it is highly likely that many paths in the program although remaining feasible are never executed. Models generated from code can be very large, in an experiment using Bandera [3], generating a model from 37 lines of Java code produces more than 200,000 lines of model code [15].

Models extracted from traces included runtime information that would otherwise be unavailable from code. Due to inheritance and polymorphism, actual methods called at times can only be known during runtime. Of course, in most cases, values of parameters passed to a method can only be known during runtime. Aliasing is a hard problem to solve with static analysis; however, information on which address a pointer points to is available during runtime.

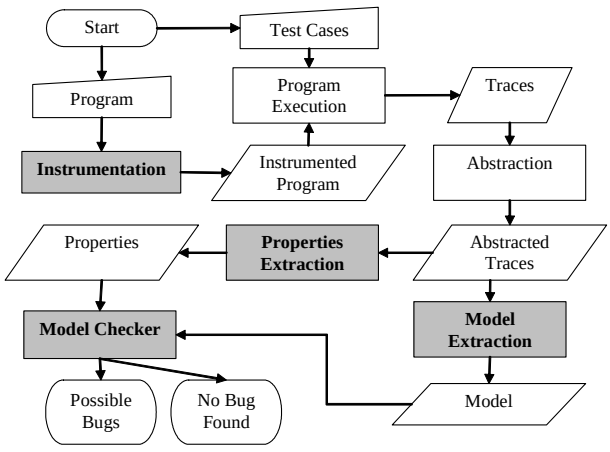

Figure 1. Proposed Mining-Model Checking Framework

One can also find bugs by comparing the traces themselves to the mined properties. However, the model learned from the traces is preferable as it can be used to further characterize the context where a bug occurs. Hence, not only the buggy trace but also the set of other potentially buggy traces can be revealed.

4. Conclusion and Future Work

In this preliminary study, we have proposed a novel framework that allows model checking to be performed in the absence of code, model and properties. Both the model and the set of properties are extracted by learning an automata model and a set of strong LTL properties from traces. A novel end-to-end framework starting from a program to the identification of potential bugs have been proposed. In [17, 18], we have shown case studies and experiments on mining models from traces. In [16, 19], we have shown case studies and experiments on mining bug-revealing properties from traces. Due to the space limitation, we refer interested readers to the above papers for more details. We plan to investigate the practicality of this unique approach to model checking through further case studies and experimentation, and through improving the accuracy of learning models and mining more complex temporal properties.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *POPL*, pages 4–16, 2002.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[3] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H.-J. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, 2000.

[4] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of Symp. on Operating Systems Principles*, 2001.

[6] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transaction on Software Engineering*, 27(2):99–123, February 2001.

[7] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *IFM*, 2005.

[8] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, 2006.

[9] G. Holtzmann. The logic of bugs. In *SIGSOFT FSE*, 2002.

[10] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.

[11] JRAT. Java runtime analysis toolkit. *online at http://jrat.sourceforge.net/*.

[12] J. R. Larus and E. Schnarry. EEL: Machine-independent executable editing. In *Proc. of SIGPLAN Conf. on Programming Language Design and Implementation*, 1995.

[13] M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. Academic Press, 1985.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transaction on Software Engineering*, 32(3):176–192, March 2006.

[15] D. Lo. Projects:open discussion. http://projects.cis.ksu.edu/forum/forum.php?thread_id=665&forum_id=4, 2007.

[16] D. Lo and S.-C. K. (Advisor). A sound and complete specification miner. In *SIGPLAN PLDI Student Research Competition (awarded 2^{nd} position) – www.acm.org/src/winners.html*, 2007.

[17] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.

[18] D. Lo and S.-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *FSE*, 2006.

[19] D. Lo, S.-C. Khoo, and C. Liu. Automatic extraction of temporal rules for model checking. In *VMCAI (under submission)*, 2007.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[21] L. Mariani, S. Papagiannakis, and M. Pezze. Compatibility and regression testing of COTS-component-based software. In *ICSE*, 2007.

[22] B. Morse. A c/c++ front end for the daikon dynamic invariant detection system. *Master's Thesis, MIT*, 2002.

[23] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[24] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.