Software Verification Through Theorem Proving and Model Checking

Advancing the state of correctness

Nathan Hillyer
Computer Science
Lewis University
Romeoville, Illinois

***Abstract*—This paper covers software verification and its applications. Software verification is a formal way to prove a program is correctly matching a specification. This differs from testing an application, as testing doesn't prove an application works; rather, it tries to find ways in which the program does not work. Software verification can also be distinguished from debugging, because debugging is a way to fix an application that has been shown to not meet its specification, commonly through some form of testing. Theoretically, if a software has been verified to fully match its specifications, there is no need for further testing or debugging. The art is figuring out how to draft a formal specification and what tools we can use to implement verification in the software development process. We will focus on two methods of software verification: Theorem proving, and model checking. Theorem proving is primarily distinguished from model checking by it's method of proving through formal mathematics. In contrast, model checking tests software behavior more directly instead of relying on mathematical proofs. Both methods have their strengths and weakness: Theorem proving is more rigorous but requires specialized knowledge and is time-consuming, whereas model checking is able to test many states with a single definition, but can have problematic performance issues when used within a problem domain that has a large or infinite number of possible states.**

Keywords—formal verification methods; software verification; testing; theorem proving; model checking

I. INTRODUCTION

Formal verification is a mathematical method of testing software systems. To do this, a user of formal verification creates a formal model of a system's behavior. Then the claim, or correctness, of this system is defined in a formal specification language that specifies exactly how the system is expected to behave. Based on the model of the system and the specification of how it should work, a formal verification method then demonstrates whether or not the model is correctly solving the problem [1].

There are generally considered to be three methods of formal verification: Theorem proving, Model checking, and static analysis. Theorem proving uses deduction to create a formal mathematical proof through inductive arguments and proofs supplied by humans. Model checking explores the ways a program can be executed and makes sure it meets expected behavior through humans creating a model of the systems along with specifying how it works. Static analysis automates program execution and warns the user if it violates any rules within the static analyzer [2].

This paper addresses theorem proving and model checking, as these are two fledgling areas of computer science research. Static analysis has been implemented in most modern IDEs, whereas theorem proving and model checking have been largely restricted to mission critical systems and academia. This may be seen as surprising, at first glance, because out of the three, static analysis can be seen as the least rigorous, and perhaps the least useful from a standpoint of proving a system is correct. However, static analysis has a broad applicability, where rules can be applied to all programs without the need for human intervention, so it is the author's opinion that this is why static analysis is more common in day-to-day software development.

Theorem proving involves the use of a theorem prover to prove that a component of a hardware or software system is working correctly. There is very little difference between proving mathematics and proving the correctness of a system, because theorem proving involves describing hardware and software in mathematical terms. Theorem provers attempt to automate as much of this process as possible, but humans are still expected to provide the bulk of the work [3].

Model checking is an attempt to provide formal validation of specifications related to an information system. It attempts to bridge the gap between the ease of testing and the difficulty of theorem proving—as it automates more of the correctness process than theorem provers. It is most applicable to engineering practices that provide the bulk of their implementation through models or specifications. In such cases, as long as the algorithms are correct, the system checker only needs to validate the models. There are many tools that have implemented model checking [4].

II. AUTOMATED THEOREM PROVING

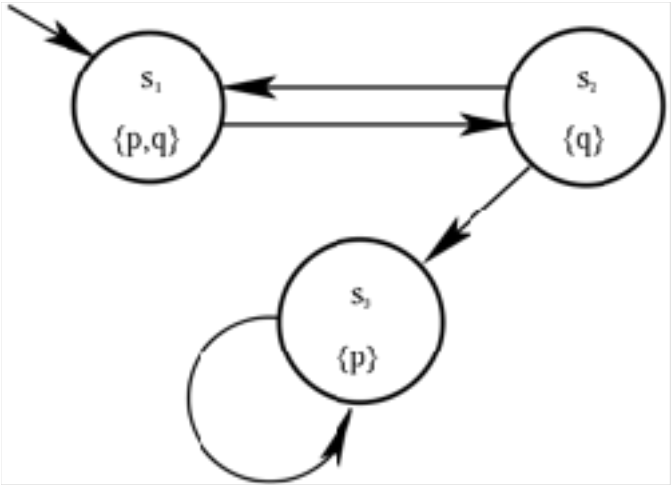
Automated theorem proving formally appeared in 1957, with the paper entitled, “Empirical explorations of the logic theory machine: a case study in heuristic” by Newell, Shaw and Simon [5]. They presented a program called Logic Theory Machine, which was created in order to learn how to solve difficult problems such as proving mathematical theorems [6]. However, the roots can also be traced to Martin Davis’ programming of an algorithm that proved the sum of two even numbers is even [7].

There are two forms of automated theorem proving: Fully automatic proof searches and interactive proof searches. Interactive proof searching involves the editing of a proof where a human guides the search and combination of existing proofs programmed into the computer system.

Theorem proving, in its essence, contains three components: The precondition, the program (either an entire program or a component thereof), and the post condition. Axioms, rules, existing proofs, and logical deduction are used to find the result, or postcondition, based on the input of a precondition to a program. This is usually specified as formal logic formulas. Because theorem proving lays claim within the realm of pure logic, it does not need to enumerate all program states in order to verify the correctness of some components. The disadvantage of this approach is there must be a knowledgeable user guiding this process and translating, if needed, modified programs fitting the pre and post condition translations back into the software system. Despite this disadvantage in the amount of human labor required, theorem proving has the advantage of showing the complete absence of errors without having to run through each program state [1].

III. MODEL CHECKING

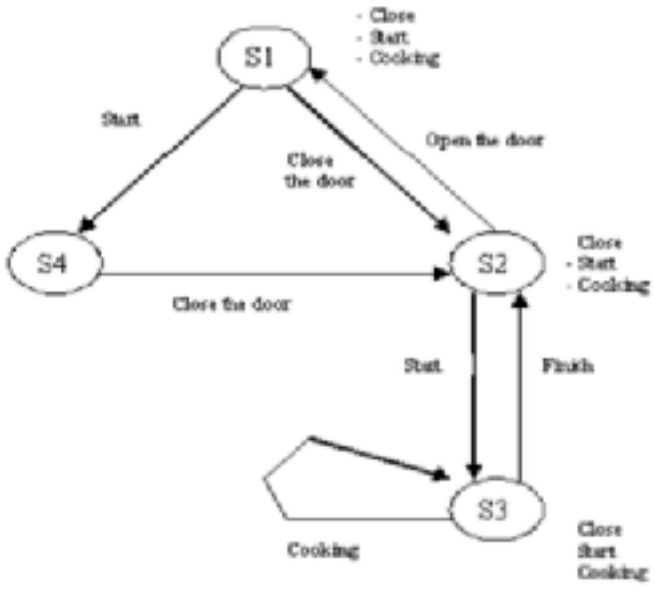
Model checking is a way of testing the states of a system in order to prove whether a model fulfills a certain specification. The specification is in the form of a logical formula and the model is described with a Kripke structure, otherwise known as a labeled state-transition graph [8].



Example Kripke structure

Model checking utilizes a modeling language that can describe the system and through the modeling language a model checking tool can explore different possible states to see if the system behaves as expected. Modeling languages work with the skill of temporal logic [1]. Temporal logic is a form of logic where the system and rules are specified according to time, such as: “Every time when a message is sent, an acknowledgment of receipt will eventually be returned and the message will not be marked ‘sent’ before an acknowledgment of receipt is returned” [9]. Model checking can also be defined more formally as “Given a model M and a formula ϕ, model checking is the problem of verifying whether or not ϕ is true in M” [10].

Model checking is composed of three activities. First, one defines the formal model of the system that one wishes to verify within the modeling language of the model checking tool. Second, one comes up with a property of the system that they’d like to prove, or, in other words, a question of the system’s behavior that the model checker can solve [1]. Here is an example of a microwave oven system that may be modeled [11]:



IV. Model Checking Tools

There are many model checking tools available to the modern computer scientist. These tools largely come in four categories. There are explicit state model checkers that use an explicit representation of the system with a model specification. Sometimes this transition system is computed prior to verifying the properties of the system, otherwise they are computed on the fly while verification is being performed. Symbolic model checkers, on the other hand, represent the transition system as a formula in Boolean logic. Bounded model checked consider traces, with a maximum length, of the transition system represented by a Boolean logic formula. The final category, constraint satisfaction model checkers, use logical programming to verify formulae [4].

A. Spin

Spin is the original model checker developed in 1980. It is an explicit state model checker that computes the transition system on the fly. This has the advantage of not constructing a global state transition graph, but the disadvantage is that each transition must be recomputed for each property. Spin uses the Promela model specification language, an imperative language with concurrency support inspired by the C programming language. It then uses Linear Temporal Logic formulae and sees if it holds up through every possible state of the Promela model [4].

B. NuSMV

NuSMV is one of the first model checkers belonging to the symbolic model checking category. NuSMV utilizes a language called SMV to illustrate finite state machines. System transitions are represent by transition constraints given by a Boolean formula that restricts the set of next values. Compared to Spin's Promela language, SMV specifications can be longer because the language has lower level abstractions.

C. FDR2

FDR2 is a model checker belonging to the explicit state modeler category, much like Spin. It can also be considered an implicit state model checked due to it's ability to compress the state-transition graph as it's building it and checking properties. FDR2 uses a variant of communicating sequential processes, called CSP_M.

D. CADP

CADP is a toolkit which includes model checking components. The model checking components use explicit state model checking. CADP uses a temporal logic meta-language called XTL for it's model checking.

E. Alloy

Alloy is a model checker that falls in the symbolic model checking category. It uses a first-order logic language that has relations in its style of terms. An Alloy specification will have a set of signatures that define sets and relations. There are also constraints that apply conditions to these signatures.

F. ProB

ProB is a constraint satisfaction model checker. It supports a variety of languages, but for our purposes we will evaluate the B language implementation where the specifications are organized into abstract machines. ProB properties can be written in computation tree logic and linear temporal logic.

V. Practical Applications In The Mobile Development Industry

Mobile development is often a bug-ridden process where companies have to sacrifice software quality in order to be the first or leading application to enter a particular market. Once an application has its initial release, it can often be difficult to test and debug all the potential states within the system. A robust model checker designed for mobile development paradigms could be a welcome addition to the software development process. Indeed, there seems to be progress on this front with a model checker, Java PathFinder by NASA, being converted to Android [12].

JPF-Android, as the project is called, has been going steady for 5 years now and is now at 950 git commits—a fairly active project. There doesn't appear to be widespread adoption, as of yet, but this could change as the project continues to mature.

Whether it's JPF-Android or another project, it is this author's opinion that the rigor offered by model checking will eventually be used for mobile development. The promise of being able to have some proof that an application is bug-free, or at least free of bugs within a specification, is too much to resist.

References

1. T. Reinbacher, "Introduction to Embedded Software Verification," *Vienna: University of Applied Sciences Technikum Wien,* 2008.
2. P. Cousot and R. Cousot, "A gentle introduction to formal verification of computer systems by abstract interpretation," *Logics and Languages for Reliability and Security,* vol. 25, pp. 1-29, 2010.
3. J. Avigad, L. de Moura, and S. Kong, "Theorem Proving in Lean," 2016.
4. M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," pp. 581-596: Springer, 2010.
5. W. W. Bledsoe, Automated theorem proving: After 25 years. pp. 1-3: American Mathematical Soc., 1984.
6. A. Newell, J. C. Shaw, and H. A. Simon, "Empirical explorations of the logic theory machine: a case study in heuristic," pp. 218-230: ACM, 1957.
7. A. J. A. Robinson and A. Voronkov, *Handbook of automated reasoning*. Elsevier, 2001.
8. E. Clarke, "The birth of model checking," *25 Years of Model Checking,* pp. 1-26, 2008.
9. V. a. G. Goranko, Antony, The Stanford Encyclopedia of Philosophy, E. N. Zalta, ed., Winter 2015 ed.: Metaphysics Research Lab, Stanford University, 2015. [Online]. Available: https://plato.stanford.edu/archives/win2015/entries/logic-temporal/. Accessed on 2017-04-19.
10. F. Raimondi, "Computational Tree Logic and Model Checking A simple introduction," *Validation and Verification Course Notes, University College London,* 2007.
11. B. Nixon, L. Chung, J. Mylopoulos, D. Lauzon, A. Borgida, and M. Stanley, "Implementation of a compiler for a semantic data model: Experiences with taxis," *ACM SIGMOD Record,* vol. 16, no. 3, pp. 118-131, 1987.
12. H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using Java PathFinder," *ACM SIGSOFT Software Engineering Notes,* vol. 37, no. 6, pp. 1-5, 2012.