Model Checking as Error Correction

A Survey of Important Concepts, Topics, and Applications of Model Checking

Devin Gonier
Department of Computer Science
Lewis University
Romeoville, IL
DevinCGonier@lewisu.edu

***Abstract*—Model checking is a method for finding errors by analyzing states and transitions of software. The goals of model checking should be based on appropriate expectations that factor in elements of complexity and efficiency. After a brief overview of how model checking works in a basic sense, its goals and challenges are explored. Then, different techniques for model checking are discussed, followed by examples of applications for model checking. The paper concludes with a look into the future of model checking.**

Keywords—Model Checking, State-Space Explosion, Embedded System Model Checking, Intelligent System Model Checking, Explicit State Model Checking, Symbolic State Model Checking, Bounded Model Checking, Constraint Based Model Checking.

INTRODUCTION

Alexander Pope famously wrote, "To err is human; to forgive divine" [7]. Indeed, forgiveness is difficult when simple mistakes that could have easily been avoided cost great fortunes or cause terrible tragedies [6]. From 1985 to 1987 six people died from receiving too much radiation during treatments. The cause of the improper and devastating treatment was a software bug in the Therac-25 radiation program [5]. In 1999, the Mars Climate Orbiter, a $327.6 million project, was brought down by a simple failure in the "translation of English units into metric units" [4]. Of course, in each of these cases parties involved were not so negligent as to skip the testing and bug checking part of their work. Nor, was the true error in the original mistake, because such mistakes are inevitable. Rather, the error came during the process of checking the software for bugs afterwards. Error correcting is as important as it is complex. In other words, error correction is tricky, because it is itself prone to oversights and errors. This paper provides an overview of a process called "model checking", its goals, types, and applications. Section one outlines some of the basics of model checking and its challenges. Section two provides a brief overview of different model checking techniques. Finally, section three discusses applications and concludes with a comment on the future of model checking.

I. GOALS & CHALLENGES OF MODEL CHECKING

A. Model Checking Basics

Model checking at its most basic level refers to software designed to check other software for mistakes. More specifically, model checkers explore the different possible states and transitions of a program and test those states and transitions according to given propositions or theorems. When propositions fail, counterexamples are provided for programmers to examine and correct. A *state* refers to an "evaluation of the program counter, the values of all program variables, and the configuration of the stack and the heap." A *transition* "describes how a program evolves from one state to another." [2] It's important to clarify and qualify the ambition of checking a program's states and transitions in a few ways.

Model checkers provide a "guarantee of quality," which usually does not mean "total correctness of a design," since to do so with complicated software would often end up being far too expensive. To further break this down consider the distinction between *debugging*, *verification*, and *testing*. [2] *Debugging* is "the process of diagnosing the precise nature of a known error and correcting it afterwards" [8]. *Verification* differs from debugging in that one attempts to prove that a program satisfies a proposition or specification. *Functional* and *formal* verification are two types of verification which were "almost synonymous." Both refer to a systematic, mathematical, and logical approach to proving that a system operates correctly. Functional verification is an "elusive goal" because to prove that a program functions perfectly requires a level of specification in the beginning that is very challenging for highly sophisticated programs. Therefore, being able to functionally verify something for complex software is extremely expensive. [1] Unlike verification, *testing* refers to the process of finding counterexamples or instances where a program fails to meet a specification. [8] Different model checkers can be associated with all three activities, and are designed to efficiently find bugs and counter-examples to specifications. Usually, proving that a system operates correctly and completely lacks bugs is too difficult with sophisticated programs, thus the goal usually becomes "finding bugs instead of proving their absence" [1].

Model checkers typically focus on two main types of problems: *liveness* and *safety*. All programs involve different steps, and at each step the program is in a particular state where functions are being carried out. A program is "live" if it can progress along an appropriate path of states. It is no longer alive if it becomes stuck in one particular state or if it reaches a point of infinitely looping through the same set of states. Additionally, *liveness* can refer to the fact that "sufficient

conditions" exist that "enable an action." In other words, certain actions are allowed to happen given the appropriate input or conditions. Model checkers are designed to identify states that breach liveness criteria when they occur and demonstrate how they occurred in the first place. [3]

Safety constraints on the other hand are almost the opposite of liveness constraints in that these constraints are designed to ensure certain actions are not performed by users. These constraints specify "necessary conditions" for taking an action, and therefore prevent users from doing things they are not supposed to do. This could include procedures that prevent library members from checking out books that are reserved for someone else, or preventing users from accessing information in a network they are not allowed to access. Typically, these constraints are specified and then tested by model checkers to see if there are vulnerabilities which would allow users to do inappropriate things. [3]

As was previously mentioned, model checkers are usually unable to check every single aspect or potentiality of a piece of software, but instead operate as efficiently as possible to create a "guarantee of quality." The qualifications of that guarantee are ultimately based on whether the software violates a series of propositions initially specified. These propositions are usually carefully constructed logical conditions and constraints for the transitions and states of the program. It is challenging to create a sufficiently extensive set of propositions to feel confident with the results, which is why this is viewed by some as a "bottleneck" to model checking [1]. The propositions are then systematically evaluated in various states and transitions via the most efficient means. Usually, this means the model checker hunts for counter-examples to the initial constraints and conditions in different states and transitions or their sets.

Counter-examples are extremely useful for checking real-world systems as they point to flaws that would otherwise be easily overlooked. Since it is difficult or impossible to understand every conceivable way a user may use a system (i.e. all the different parameter choices from beginning to end), unexpected scenarios may reveal the need for further specification of rules or recoding of processes. Analyzing the variety of possible scenarios can become exponentially complex, which is why logics are often used in different ways to efficiently evaluate transitions and states. The problem of having too many states and transitions is often referred to as *state-space explosion*.

B. The Challenge of State-Space Explosion

For any single user experience, a series of processes occurs which are determined by choices and input made by that user. While some choices may be simple Boolean options, most of the time users are able to input scaled values which could have a wide range of consequences. The input from the user is then typically passed to functions that use that input in variables. As a result, if algorithms are not carefully crafted, the right user input may result in an error like *divide by zero*.

Instead of iterating through a potentially infinite set of values, model checkers use *abstraction* methods to limit the number of potential inputs. For example, if an algorithm depends on two variables *(a, b)*, then the model checker can check for scenarios in which $a<b$ or $a>b$, or $a=b$ instead of iterating through specific numeric assignments of the variables (i.e. a=1, b=4; a=2, b=3, and so on). Furthermore, the inputs can also be broken down into abstract domains, where numbers can be further differentiated by concepts like signs (positive, negative, or zero) and then relations between signs and values, such as b is a negative number less than another negative number a. By iterating through abstract domains and relationships, model checkers can more efficiently find errors, such as the error *divide by zero* when $x=y$ in an algorithm that employs the equation $1/(x-y)$. [2]

There are many other methods for limiting the state-space explosion phenomenon. For example, one can use symbolic methods to create different sets of states and transitions that share the same types of properties or relations. Propositional and temporal logic can be used to model states and transitions as sets in addition to their use in specifications and testing. Instead of iterating through each state or transition individually, by associating them in sets, one can determine the validity of large groups in discrete evaluations.

Another category of state-space reduction is called *decomposition*, in which complexity is reduced by subdividing "the verification target into components so that their properties are small enough to be verified separately [1]. An example of this is a procedure called *partial order reduction* to determine whether the order of operations plays a role on the quality of a states [2]. If order of previous states and transitions doesn't matter, the state or transition in question can be evaluated independently of the flow of events leading to it. One could even realize redundancies in the process flows of certain transitions and states and then bypass those redundancies, taking shortcuts, to certain states. These methods help reduce the computational challenges associated with modeling all the different states and transitions in a program and therefore minimize the effects of state-space explosion. Different model checkers of course employ different combinations of techniques. There are many more techniques than those outlined here to reduce state-space explosion (like Buechi automata or modularization). However, other more general strategies to model checking exist which aren't just about reducing the effects of state-space explosion and relate to the effectiveness of catching errors [1]. Section two explores some primary categories of these techniques.

II. TYPES OF MODEL CHECKING

The first of the four primary types of model checkers is the *explicit state* model checker. This approach depends on an explicitly (as opposed to symbolically) defined map of the various states and transitions. This mapping can either happen prior to the actual checking process or *on the fly* as the checking is going on. Examples of systems that use explicit state model checking are CADP, SPIN, and FDR2 [3].

The second approach is *symbolic state* model checking. As the name suggests, states and transitions are not explicitly graphed but are symbolically grouped and relationally associated using propositional logic. This allows for logical

evaluations of multiple states and transitions simultaneously, and can have benefits in reducing state-space explosion. One common type of propositional grouping can be the constraints associated with a particular state or transition. An example of a symbolic state model checker is NuSMV. [3]

A third approach is *bounded* model checking. Bounded checkers evaluate the graph of states up to a certain number of steps (typically denoted with the variable *k*) or bound the "domain of system variables." This process yields results that hold up only to a chosen bound and can help with narrowing or honing in on specific types of problems [1].The process can be repeated for different values of k, allowing one to eventually rule out different violations. NuSMV and ALLOY are examples of systems that use bounded approaches.

Finally, *constraint satisfaction* model checkers specify constraints that must be satisfied at different states and transitions. The checker evaluates different states and transitions according to specified limitations and looks for instances where constraints are not satisfied as counter-examples. For example, games like sudoku operate according to certain constraints: a number cannot be used twice in a given row, box, or column. If a particular state violates a constraint, the counter-example is flagged. An example of a constraint satisfaction model checker is ProB.

As should already be clear from the examples given, these types of model checkers are not necessarily mutually exclusive. Model checkers can be constraint based and symbolic state and so forth. Different approaches attempt to balance and leverage the various advantages associated with these different techniques to create a more efficient evaluation of the different states and transitions in a program.

III. Applications of Model Checking

Model checkers are used for a variety of different programs. They are crucial in checking programs that progress through stages based on user or environmental input. Model checkers like ProB or NuSMV are used in information systems all the time to evaluate how users can and should interact with information. Can users always access the information they should be able to access and not the information they shouldn't be able to access? How should information be stored in a system and how should it be changed? Since these systems are typically very dynamic, it is crucial that they are well designed from the beginning as errors can cause complex systems to crash or expose vulnerabilities to parties that would exploit them.

Embedded systems are another kind of system which is often evaluated using model checkers. Embedded systems operate machines which perform functions in an environment. Such systems can be as simple as a watch, or as complicated as a vehicle. In embedded systems, environmental influence or user input can determine how the machine operates relative to its environment, and are therefore susceptible to the state-space explosion phenomenon. Since many machines like traffic lights or manufacturing machinery cannot only have financial impacts but also result in injury or death, it is essential to use modal checkers to evaluate the types of scenarios that may arise. In this case safety constraints are especially important as they can prevent exploitation by outside users or dangerous activities that would otherwise be prevented with the right constraints on sensory input.

It is interesting to imagine other applications for model checkers in systems that utilize machine learning algorithms. One can imagine advanced intelligent systems that are able to write and revise their own source code. This process is sometimes referred to as bootstrapping when the writing of code improves the entity writing it. This creates a feedback loop in which the machine can become increasingly better with each new iteration of coding. Bootstrapping without appropriate model checking could be disastrous. Any new code ought to conform to the same safety parameters as the original source code, and ought not to create new errors which may endanger the original source code. To successfully evaluate such code as safe seems almost as complicated as to build a system which can revise itself in this way. Thus, continued progress in this field will have important effects.

Conclusion

Model checking is important in today's society for minimizing risk associated with new technologies. Companies need to feel confident their software will operate correctly in as many circumstances as possible. In some cases, this can even mean the difference between life and death. As society becomes increasingly automated and interactions between humans and technology more complex, the vulnerability for danger grows. The more complex software becomes, the harder it is to check. The more integrated it becomes, the more dangerous it is when it fails. Furthermore, incidents of sufficient calamity could bring a halt to industry development or tighter regulations that may stunt industries from developing. Therefore, it is essential that progress in model checking technologies continues, as it is a foundation for the development of many other types of technologies.

References

[1] Beckert, Bernhard, and Reiner Hahnle. "Reasoning and Verification: State of the Art and Current Trends." IEEE Intelligent Systems 29.1 (2014): 20-29. Web.

[2] D'silva, Vijay, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27.7 (2008): 1165-178. Web.

[3] Frappier, Marc, Benoit Fraikin, Romain Chossart, Raphael `. Chane-Yack-Fa, and Mohammed Ouenzar. "Comparison of Model Checking Tools for Information Systems." GRIL (2010): n. pag. Web. 10 Apr. 2017.

[4] Isbell, Douglas, and Don Savage. "Mars Climate Orbiter Failure Board Release Report, Numerous Nasa Actions Underway in Response." NASA. NASA, 10 Nov. 1999. Web. 10 Apr. 2017.

[5] Leveson, N.g., and C.s. Turner. "An Investigation of the Therac-25 Accidents." Computer 26.7 (1993): 18-41. Web.

[6] N. Dershowitz, *Software horror Stores.* [Online] Available: http://www.cs.tau.ac.il/~nachumd/verify/horror.html

[7] Pope, Alexander. An Essay on Criticism.: By Alexander Pope, Esq. London: Printed for T. Daniel, W. Thompson, and J. Steele, and A. Todd, 1758. Print.

[8] Reinbacher, Thomas, BSc. "Introduction to Embedded Software Verification." University of Applied Science Technikum Wien (2008): n. pag. Web. 10 Apr. 2017.