available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

Linear Temporal Logic Symbolic Model Checking

Kristin Y. Rozier

NASA Ames Research Center, Moffett Field, CA 94035, USA

ARTICLE INFO

Article history:
Received 11 February 2010
Received in revised form
26 June 2010
Accepted 29 June 2010

Keywords:
Linear Temporal Logic (LTL)
Symbolic Model Checking (SMC)
Verification
Formal Methods

ABSTRACT

We are seeing an increased push in the use of formal verification techniques in safety-critical software and hardware in practice. Formal verification has been successfully used to verify systems such as air traffic control, airplane separation assurance, autopilot, CPU designs, life-support systems, medical equipment (such as devices which administer radiation), and many other systems which ensure human safety. This survey provides a perspective on the formal verification technique of linear temporal logic (LTL) symbolic model checking, from its history and evolution leading up to the state-of-the-art. We unify research from 1977 to 2009, providing a complete end-to-end analysis embracing a users' perspective by applying each step to a real-life aerospace example. We include an in-depth examination of the algorithms underlying the symbolic model-checking procedure, show proofs of important theorems, and point to directions of ongoing research. The primary focus is on model checking using LTL specifications, though other approaches are briefly discussed and compared to using LTL.

Published by Elsevier Inc.

1. Introduction

Verification of a software or hardware system involves checking whether the system in question behaves as it was designed to behave. *Design Validation* involves checking whether a system design satisfies the system requirements. (If it does not, it is desirable to find out early in the design process!) Both of these tasks, system verification and design validation, can be accomplished thoroughly and reliably using *formal methods*, such as *model checking*. Model checking is the formal process through which a desired behavioral property (the specification) is verified to hold for a given system (the model) via an exhaustive enumeration (either explicit or symbolic) of all of the reachable system states and the behaviors that cause the system to transition between them. If the specification is found not to hold in all system executions, a *counterexample* is produced, consisting of a trace of the model from a start state to an error state in which the specification is violated, providing a very helpful tool for debugging the system design.

The time-honored techniques of simulation and testing, both of which involve checking the system's behavior on a large set of expected inputs, also address similar questions and are extremely useful debugging tools in early stages of system design and verification. However, as a system is refined, the remaining bugs become fewer and more subtle and require more time to uncover. A major gap in the process of using simulation and/or testing for verification and validation is that there is no way to tell when these techniques are finished (i.e. when all of the bugs in the system have been found). In other words, testing and simulation can be used to demonstrate the presence of bugs but not the absence of bugs. There is not even an accurate way of estimating how many bugs remain. Another open question is that of coverage, of both the possible system inputs and the system state space. Quite simply, it has been proven

E-mail address: Kristin.Y.Rozier@nasa.gov.

1574-0137/$ - see front matter. Published by Elsevier Inc.
doi:10.1016/j.cosrev.2010.06.002

that testing and simulation cannot be used to guarantee an ultra-high level of reliability within any realistic period of time [1]. For some systems, this is an acceptable risk. For those systems, it is enough to reduce the bug level below a certain measurable tolerance, for example in terms of frequency in time. For *safety-critical* systems, or other systems, such as financial systems, where reliability is key because failure is potentially catastrophic, we require an absolute assurance that the system follows its specification via an examination of all possible behaviors, including those that are unexpected or unintended. This assurance is provided by model checking.

While there are a range of different techniques for formal verification, model checking is particularly well-suited for the automated verification of finite-state systems, both for software and for hardware. Once the system model and specification have been determined, the performance of the model checking step is often very fast, frequently completing within minutes. The counterexample returned in the case a bug is found provides necessary diagnostic feedback. Furthermore, iterative refinement and re-checking of the failed specification can provide a wealth of insight into the detected faulty system behavior. Model checking lends itself to integration into industrial design life-cycles as the learning curve is quite shallow and easily outweighed by the advantages of early fault detection. The required levels of user interaction and specialized expertise needed to effectively utilize a model checker are minimal compared to other methods of formal verification. Moreover, partial specifications can be checked, allowing verification steps to occur intermittently throughout system design. However, there is a trade-off between the high level of automation provided by model checking and the high level of expressiveness and control that may be required for verification in some cases. For this reason, certain systems benefit from the use of alternative verification techniques, such as theorem proving, which involves logically deducing the specification from the formal system description and a set of axioms and inference rules. Still, model checking's high level of automation makes it a preferable verification method where applicable since the performance time and quality of insight obtained from a negative result when using theorem proving for verification are highly dependent on the particular skill-set of the person providing the proof.

Formally, the technique of model checking checks that a system, starting at a start state, models a specification. Let M be a state-transition graph (i.e. an automaton) representing the system with set of states S and let $s \in S$ be the start state. Let φ be the specification in temporal logic. We check that $M,\ s \models \varphi$. In other words, we check that M satisfies ("models") φ. This technique of temporal logic model checking was developed independently by Clarke and Emerson [2] in the United States and Quielle and Sifakis [3] in France in 1981. Thus, 1981 is considered the birth year of model checking.

The primary focus of this paper is on model checking using Linear Temporal Logic (LTL) specifications. LTL was first introduced as a vehicle for reasoning about concurrent programs by Pnueli in 1977 [4]. LTL model checkers follow the automata-theoretic approach [5], in which the complemented LTL specification $\neg\varphi$ is translated to a Büchi automaton,[1] $A_{\neg\varphi}$, which is a finite automaton on infinite words that accepts exactly all computations that satisfy the formula $\neg\varphi$. $A_{\neg\varphi}$ is then composed with the model M under verification, forming $A_{M,\ \neg\varphi}$ [6]. Intuitively, any accepting path in $A_{M,\ \neg\varphi}$ represents a case where the system M allows a behavior that violates the specification φ. The model checker then searches for such a trace of the model that is accepted by the automaton $A_{M,\ \neg\varphi}$ via a nonemptiness check. If an accepting trace is found, it is returned as a counterexample. If no such trace exists (i.e. the language $\mathscr{L}(A_{M,\ \neg\varphi}) = \emptyset$), we have proven that $M,\ s \models \varphi$. This process is summarized in Table 1. The equivalent to the automata-theoretic approach for branching temporal logics utilizes automata on infinite trees and relies upon a reduction of satisfiability to the nonemptiness problem for these automata [7].

LTL model checkers can be classified as *explicit* or *symbolic*. Explicit model checkers, such as SPIN[2] [8] and SPOT[3] [9], construct the state-space of the model explicitly and create the automaton $A_{M,\ \neg\varphi}$ such that $\mathscr{L}(A_{M,\ \neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$, and $|A_{M,\ \neg\varphi}| = \mathcal{O}(|M| \cdot |A_{\neg\varphi}|)$, where vertical bars indicate the size of an automaton in terms of number of states and transitions. Next, the model checker searches for a trace falsifying the specification. This search equates to a nonemptiness check of the automaton $A_{M,\ \neg\varphi}$. The standard algorithm for this task is Tarjan's depth-first search algorithm for finding strongly connected components in the state-transition graph, which runs in time linear in the sum of the number of states and transitions. (In practice slightly more efficient algorithms are usually implemented [10–12].) However, constructing and searching the state space in this manner requires a considerable amount of space, even when utilizing optimization techniques such as on-the-fly state space construction [13–15]. Given that the size of the state space required for model checking is the largest challenge to its efficacy as a verification technique, utilizing techniques that conserve space is vital.

The *state explosion problem* is widely agreed to be the most formidable challenge facing the application of model checking to large and complex real-world systems. In short, the number of states required to model concurrent systems grows exponentially with the number of system components, constituting the main practical limitation of model checking. Sequential hardware circuits with n input variables and k registers require 2^{n+k} states to represent. Even simple systems, like an n-bit binary counter, can necessitate large state spaces (in this case, 2^n states). In general, a system with n variables over a domain of k possible values requires at least k^n states in the model and reasoning over real-valued variables, which have infinite possible values, results in a state-transition model with infinitely many states. Unfortunately, the state explosion problem is unavoidable in the worst case. However, a host of techniques have been developed over the last three decades that have successfully eased the problem for certain types of systems. For example, sophisticated data structures, clever algorithms for representing interleaving of concurrent components (called partial order reduction [16]), and the use of bisimulation equivalences [17] and compositional (also

[1] Büchi automata are formally defined in Section 3.4.

[2] http://spinroot.com/.

[3] http://spot.lip6.fr/.

Download English Version:

https://daneshyari.com/en/article/470207

Download Persian Version:

https://daneshyari.com/article/470207

Daneshyari.com