Embedded Software Verification

Complexity Begets Complexity

Brian Dutremble
CPSC 59700
Lewis University
Romeoville, IL, USA
Brian.dutremble@lewisu.edu

***Abstract*— For every increase in the complexity of software there is a corresponding increase in the complexity of verification. Verification is extremely resource-intensive but far outperforms debugging and testing when attempting to evaluate the performance of software. There are many different model checker tools, all with different strengths and weaknesses. An essential part of the verification process is choosing the correct model checker. As technology improves, model checking will become an increasingly efficient and viable method of formal verification.**

Keywords—model checking; embedded software verification; SPIN; NuSMV; FDR2; CADP; ProB; Alloy; testing; debugging

1 Introduction

As the complexity of software increases so too does the complexity of the processes by which the software can be verified. Verification is extremely resource-intensive, often consuming the majority of a project's budget, but it is the best method for determining if software is successful. Traditional methods like testing and debugging find errors through the examination of a small subset of possible system behaviors and are thus generally inadequate for increasingly complex software. As debugging and testing reach their limits, verification is showing promise for a growing number of applications.

Formal software verification consists of two main approaches: theorem proving and model checking. In theorem proving, verification is achieved through a proof of correctness through the derivation of a theorem. Theorem proving has the notable drawback of requiring interactivity that model checking does not. An experienced user must guide the tool, which can be extremely time-consuming when attempting to verify particularly complex systems. Model checking, the more popular approach, is a much easier, more reliable process for the user. Using a logical formula, model checking explores the different possible states of a system using brute force.

2 Model Checking

Model checking, which was first introduced in the 1980s, consists of three main steps. First, a model of the system must be defined in a language that fits the model checker's input language. Next, a system property to be proved – a question for the model checker to answer - must be provided. Lastly, one must use the model checker. Upon utilizing the model checker, the user will be then given a notification indicating whether the provided property was fulfilled. If the property remains unverified, a counterexample that suggests the source of the error is generated. It should be noted that one major inadequacy of model checkers their inability to deal with state space explosions. A state space explosion is likely to occur in a real life, extremely complex system. In such cases, it is sometimes best to use theorem proving.

2.1 Checker Categories

Model checkers can be divided into four major categories: explicit state, symbolic, bounded, and constraint satisfaction. Explicit state model checkers utilize an explicit representation of the transition system that can be computed prior to or during the verification process depending on the particular model checker used. Symbolic and Bounded model checkers both use a Boolean formula to represent the transition system. Lastly, constraint satisfaction model checkers use logic programming to verify a formula.

2.2 Model Checkers

There are many different model checkers used for embedded software verification but for the purposes of this analysis I will, as the research article did, focus on six: SPIN, NuSMV, FDR2, CADP, Alloy, and ProB. Some of these model checkers fit the criteria for multiple categories and most of them support temporal languages for property specification. Most model checkers are coupled with a modeling language.

2.2.1 SPIN

SPIN, one of the first model checkers developed, is tight coupled with the Promela modeling language. Specifications are written in Promela while properties are written in LTL. SPIN utilizes on-the-fly LTL model checking. This method allows the checker to avoid construction a global state transition graph but necessitates the computation of a transition for each individual property. Promela, the model specification language of SPIN, is an imperative language based in C that has the ability to handle concurrent processes.

2.2.2 NuSMV

NuSMV, a member of the Symbolic and Bounded model checker families, is based on the SMV (Symbolic Model Verifier) software and utilizes a symbolic representation of the specification to check a model against a property. NuSMV

supports the analysis of specifications expressed in both CTL and LTL and allows for the user to choose between the BDD-based symbolic model and bounded model checking. Specifications for NuSMV may me longer than in Promela because each case must be explicitly written. The language can provide a great deal of flexibility but can be inconsistent if the user is inexperienced.

2.2.3 FDR2

FDR2 is a tool of the explicit state model checker classification that uses CSP. It works by converting two CSP process expressions into labeled transition systems then ascertains whether one of those expressions is a refinement of the other. CSP supports three refinement relations: trace, stable-failure, and stable-failure-divergence. During the process refinement, FDR2 uses algorithms to reduce the size of the state-space to be explored.

2.2.4 CADP

CADP (Construction and Analysis of Distributed Processes) is a toolbox first released in 1986. It is used in many industrial projects. The property specification language of CADP is XTL. XTL features low-level operators that can be used to implement many temporal logics. All XTL formulae are evaluated on labeled transition systems. CADP has two ways of dealing with the state explosion problem that might result from its verification approaches: small models can be represented explicitly (by utilizing exhaustive verification) while larger models can be represented implicitly (by using on-the-fly verification).

2.2.5 Alloy

Alloy is a symbolic model checker. It uses first-order logic with relations as its only terms. These relations and basic sets are defined using signatures, which are similar to classes in object-oriented programming languages. Properties are written as first-order formulae.

2.2.6 ProB

ProB is a model checking and animation tool that uses the B method and B language. It is considered a constraint satisfaction model checker. Specifications are organized into abstract machines that are able to utilize substitutions to allow for generalizations and greater flexibility. ProB supports temporal languages for property specification (properties can be written in LTL or CTL).

3 Conclusion

Formal verification allows for the early detection of errors in design and thus saves resources later on in the process. While theorem proving is preferable for situations in which verification of infinite state spaces is needed, model checkers are often the correct choice, particularly when it is important for the method to be user-friendly. No matter which method is utilized, systems using software verification have been shown to be less likely to produce errors than those that do not use software verification. When utilizing model checking, choosing the correct model checker is an important part of the process, as each tool has different strengths and weaknesses.

4 Application

In my work environment, we deal with a tremendous amount of data. We are currently pouring a great deal of time and energy into making our data management processes more efficient. Software is obviously an essential part of our data management strategy and given the vastness of our databases, care must be taken when implementing any changes to the system. Our systems, thankfully, are not infinite, and I believe that they provide and opportunity for the use of model checking. We only require a relatively small number of possible outputs from our systems and the existent structure decreases the scope of any range of possible states. Currently it seems as though debugging and testing are the primary methods utilized for assessing the effectiveness of a given piece of software within our systems and I believe that model checking could have far-reaching benefits. It is my hope that our focus will shift toward formal verification, away from the current trial-and-error approach.

[1] T. Reinbacher, “Introduction to Embedded Software Verification,” University of Applied Sciences, Technikum Wien, pp. 1–11.

[2] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, M. Ouenzar, “Comparison of Model Checking Tools for Information Systems,” GRIL, Universite de Sherbooke, Quebec, Canada, June 16, 2010, pp.1-12.