Embedded Software Verification & Model Checking

(Review)

Thomas Swed
Lewis University
1 University Pkwy,
Romeoville, IL 60446
thomasdswed@lewisu.edu

Abstract— **Software Verification is used in Computer Science to assess whether a given software program satisfies all its projected requirements. It is implemented through formal verification methods such as model checking and theorem proving. This paper will discuss terminology related to software verification and validation, theorem proving and various model checkers. It will also address the application of model checkers presented in the required readings.**

Keywords—Software verification, software validation, theorem proving, model checking, debugging, testing, formal verification, temporal logic

I. Introduction

As software applications continue to grow in complexity and size, methods to verify and validate them are being sought out. While testing and debugging have traditionally been used to validate software, they do have limitations. Software verification describes the abstract process to evaluate whether a software system satisfies its intended purpose at a high level.

II. Software Verification and Validation

A. *Key Terms*[1]

- *Debugging* refers to the process of discovering and correcting errors that prevent the intended function of a software program from working.
- *Verification* aims to prove that a given program meets all expected requirements. This is often accomplished through a formal mathematical proof and seeks to answer the question: "Is this system being built in the correct manner?".
- *Validation* technically differs from verification. While verification focuses on the method or way in which a product is being created, validation determines if the system being built satisfies its intended purpose [2].
- *Testing* is used to detect software deficiencies with regard to its designed goals. While verification is used to evaluate whether specifications have been met, testing determines where a program has failed to meet those specifications [3].
- *Formal Verification* proves through the use of formal mathematical proofs the correctness of an algorithm in relation to its intended design.
- *Theorem Proving* is one of two primary approaches to formal verification. It consists of producing a compilation of mathematical proofs to prove the correctness of a system [4].
- *Model Checking* describes the second primary approach to formal software verification. This is typically the more widely used approach to automatically determine the correctness properties of a system's state. Specifications for model checking are written in propositional temporal logic.

III. Theorem Proving

Theorem proving requires a considerably greater amount of time and knowledge than the more widely used method, model checking. It utilizes mathematics and logic to accomplish the task of showing that a statement is true based on previously established statements. A characteristic that distinguishes theorem proving from model checking is that the former does not need to check a program's state in order to verify certain properties [5]. Because these properties must be verified at every step, theorem proving is a labor-intensive, manual task. This is why model checking offers a much more viable implementation.

IV. Model Checking

Model Checking is the most widely used approach to verify software systems formally. It provides broader coverage than simulation or testing and requires less human interaction than theorem proving, enabling it to be automated [6]. There are numerous tools available for model checking. Four families will be evaluated: explicit state, symbolic, bounded and constraint satisfaction model checkers.

A. *Model Checking Families*

- *Explicit state* model checkers plainly represent the transition system of a model's specification. The system is computed prior to property verification (CADP, FDR2) or on-the-fly while checking a property (SPIN) [7].

- *Symbolic* model checkers include NuSMV and represent the transition system as a Boolean formula [8].

- *Bounded* model checkers verify whether a property error has occurred in *k* or fewer steps using a Boolean formula. NuSMV is also in this family.

- *Constraint satisfaction* such as PROB are designed to find a solution using logic programming applied to a set of constraints that impose a condition.

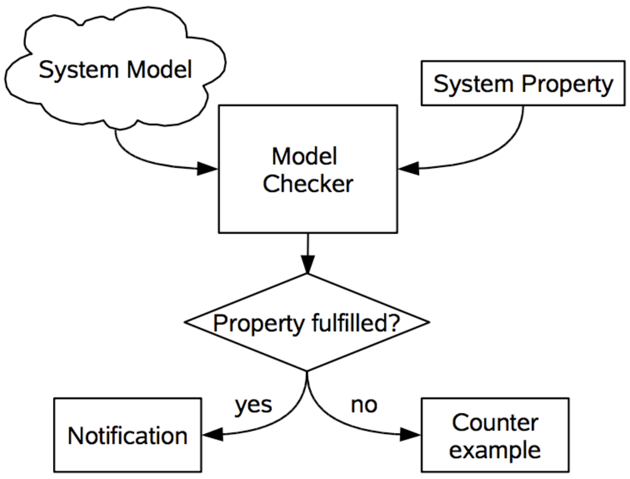

Fig. 1. A typical model-checking workflow [9].

V. Model Checking Tools

There are a number of model checking tools available; six will be evaluated.

A. SPIN

SPIN ("Simple Promela Interpreter") is a general tool for verifying the correctness of a software model using on-the-fly LTL model checking [10]. Promela is used as the descriptive language of SPIN and allows for basic data types such as *char, bit* and *int,* as well as arrays. It is built on the C programming language and expresses properties as LTL formulas. Common operators of these formulas include *always, eventually* and *until.* SPIN also does not account for events of systems, only a given state.

B. NuSMV

NuSMV is an extension of SMV or Symbolic Model Verifier software. It validates temporal logic properties in finite state systems with implicit techniques [11]. NuSMV was the first model checking tool based on Binary Decision Diagrams (BDDs) and supports the analysis of specifications in CTL or LTL. Within a specification are module declarations that include variable declarations and constraints.

C. FDR2

FDR2 doubles as an explicit and implicit state model checker specifically for communicating sequential processes (CSP). It has the ability to evaluate deadlocks, livelocks, refinement and determinacy of process expressions [12]. It is implicit in the sense that it compresses a state transition graph using state space reduction techniques to verify properties.

D. CADP

CADP contains many different tools that are used to design communication protocols and distributed systems [13]. To define temporal logic properties, CADP uses XTL which implements several different temporal logics, including HML, CTL, ACTL, and LTAC. In the article [14], LOTOS-NT was chosen as the second component of CADP to define models. It uses algebraic conditions as well as process expressions to determine specifications. Conventional algebraic operators (sequence, choice, loops, etc.), state variables and assignment statements are also supported.

E. ALLOY

ALLOY creates models using first-order logic which can in turn be automatically verified for correctness. These models are characteristically relational and use signatures to establish sets and relationships. Supported operations on relations include union, intersection, difference, join, transitive closure, domain and range restriction [15].

F. PROB

PROB uses the B-Method to produce animation and model checking. Specifications in the B language are categorized using "machines" which contain state variables, an invariant constraining the state variables and operations on the state variables [16]. PROB also combines the benefits of several languages (LTL, past LTL, CTL) to define properties.

VI. Application

A. Microcontroller Assembly

Research is currently being performed to generate model checkers directly from source code, rather than building a new system from the ground up. In the case of embedded systems, C is common as the source code to access specific microcontroller functions. However, this approach is limited in verifying microcontroller software. Assembly code model checking was explored to solve this problem. This approach allowed errors to be discovered in development that were not detected in the C code. Yet in order to build an assembly code model, a greater amount of time is needed due to its complexity and requirement for a model of the entire microcontroller. Despite the challenge of initially creating an assembly code model, once one is built, it can be used to verify all software related to the microcontroller. Another advantage is that no expert knowledge of system modeling is assumed of the user, resulting in minimal training and a tool that is simple to use [17].

B. Information Stystems Case Study

The second article [18] evaluates all six of the previously mentioned model checking tools in a case study. The purpose of the study was to evaluate and compare the performance of six model checking tools on a library system. Liveness properties are differentiated from safety properties throughout the study. The former identifies whether the system is stuck in any type of lock while the latter describe conditional requirements needed to perform an action. The main focus was on the control part of an information system (IS) which determined the sequence of actions that the IS must accept [19]. Three questions were used as a comparison for each tool [20].

1) Is the modeling language of the model checker adapted for the specification of IS models?

2) Is the property specification language adapted to specify IS properties?

3) Is the model checker capable of checking a sufficient number of instances of IS entities?

C. Results of IS Case Study

Upon comparing the six model checking tools, several prominent features were discovered [21].

- *Model specification language: abstraction over entity instances.* A feature to structure the number of instances for each entity and association.
- *Model specification language: representation of entity and association structures.* This feature model's actions involving many different associations.
- *Model specification language: representation of IS scenarios.*
- *Property specification language: abstraction over entity instances.* Used to abstract from entity instances using quantifying variables.

An efficient model checker specifically for IS must be able to perform several differing functions, including support for states and events. Process algebraic operators are preferable for their ease of use in IS scenarios while state variables are effective in simplifying property specifications. Based on the case study presented, PROB was the model checker of choice for its ability to cope with various and diverse functions effectively.

VII. Proposed Application

Model checking principles can be applied to the order entry system used in my company. As a book publisher, orders are taken over the phone for book stores among other customers. The software used to input these orders has been struggling with the demands of our clients and in-house sales team. Model checking techniques can further evaluate the condition of various states in the system to identify and eradicate errors.

VIII. Conclusion

Formal verification is gaining acceptance as a means to provide a greater amount of coverage than testing or debugging, as well as identify errors early in the development process. Theorem proving and model checking each have distinct strengths and weaknesses; which one to use depends on a user's needs. Theorem proving relies heavily on mathematical proofs while model checking verifies system properties using a modeling language. Model checking has become increasingly preferable to theorem proving for its ease of use and widespread application. The benefits of using formal verification techniques outweigh the costs as they often lead to more efficient systems with fewer errors.

References

[1] Thomas Reinbacher, Article: Introduction to Embedded Software Verification. Embedded Computing Systems Group, Vienna University of Technology, Austria, 2008, 2.

[2] "Software Verification and Validation". Wikipedia, https://en.wikipedia.org/wiki/Software_verification_and_validation. Accessed April 13, 2017.

[3] Reinbacher, 2.

[4] Reinbacher, 3.

[5] Reinbacher, 3.

[6] Mark Frappier, Benoit Fraikin, Romain Chossart, Raphael Chane-Yack-Fa, Mohammed Ouenzar. Aritcle: Comparison of model checking tools for information systems, June 2010, 6.

[7] Frappier, 8.

[8] Frappier, 8.

[9] Reinbacher, 4.

[10] "SPIN model checker". Wikipedia, https://en.wikipedia.org/wiki/SPIN_model_checker . Accessed April 14, 2017.

[11] "NuSMV". Wikipedia, https://en.wikipedia.org/wiki/NuSMV. Accessed April 14, 2017.

[12] Frappier, 10.

[13] "Construction and Analysis of Distributed Processes". Wikipedia, https://en.wikipedia.org/wiki/Construction_and_Analysis_of_Distributed_Processes. Access April 14, 2017.

[14] Frappier, 6-18.

[15] Frappier, 11.

[16] Frappier, 11.

[17] Reinbacher, 8.

[18] Frappier, 6-18.

[19] Frappier, 6.

[20] Frappier, 6.

[21] Frappier, 16-17.