Formal Verification

Joan Yoon
College of Arts and Sciences
Lewis University
Romeoville, IL 60446
Email: joanhyoon@lewisu.edu

***Abstract*—This paper presents the main themes established in Thomas Reinbacher's "Introduction to Embedded Software Verification" and Marc Frappier and associates' "Comparison of Model Checking Tools for Information Systems." These reviews concentrate on applying formal verification tools, and mainly on model checking. With analysis and learned insight from these studies, I will briefly propose application of these concepts to address software verification issues in my professional activities.**

I. Introduction

Technology is constantly growing, and continues to show us the great potential in improving all aspects of our life. In order to keep up with the advances, the backend in regards to security and updates must also align. Methods such as software testing and debugging were in place to make sure that a system functions properly. However, while these tools did help in formal verification, it had its limits in terms of reliability and difficulty handling certain levels of complexity. As a result, allegedly more efficient methods such as theorem proving and model checking was introduced.

II. Growing Past Traditions

A. Traditional Methods

The two traditional methods for verification are debugging and testing. The former is more of a resolution measure after the problem has occurred; it must then be identified and corrected. Testing is also less of a prevention tactic in that this process involves more observation of the behavior of a system in order to detect bugs and other issues. Similar to trial and error type of experiments, it is essentially a never-ending process; the product can always be improved and so it is necessary to persistently test different inputs for the best reactions [2].

In consideration of the steps to complete the verification in these basic methods, although simple in structure, it is obvious that a lot of time and effort is still necessary. Is it an efficient allocation of resources? This is a questioned reasoned with why these rather basic tools are being overlooked for more intricate designs.

B. Embedded Software Verification

Reinbacher states that "literature distinguishes two main areas of formal software verification approaches [2], which are theorem proving and model checking.

1) Theorem Proving: Theorem proving is the more mathematical method by representing the systems model and property as a logical formula. This side can verify infinite state spaces and also can focus on certain state space to verify certain properties. However, it requires expert-level user intervention and guidance for processing [2].

2) Model Checking: On the other hand, model checking looks at a systems finite-state model and a logical property to systematically check if it satisfies a certain specification. It is most commonly presented by referring to the Kripke structure, which is defined as an ordered sequence of: a finite set of states, an initial state, transition relation, and an interpretation function. Computation tree logic (CTL) is applied as a major descriptor, as it combines both linear temporal and branching-time logic to describe events. Combining these factors, model checking can provide counterexamples and even show the absence of errors. However, it must go through the entire model to verify a specific property, and it runs into difficulties when applied to a real-life system that may be larger and more complex, causing a state space explosion [2].

In the paper "Introduction to Embedded Software Verification," he briefly applies the concepts to a coffee vending machine example and to microcontroller assembly code. For the former, the functions of a coffee vending machine can be broke down in terms of the Kripke structure variables. After inserting the proper amount, a valid coffee selection is made to brew the coffee, and give change, as necessary; the option to abort is available at any time. Expanding upon application onto more technical situations, the microcontroller assembly code application is done to show that model checking tools can create a systems model directly from source code. Although this is much simpler than model checking programs in high level programming languages, a whole model of the microcontroller which are usually complex is needed to implement. Even so, the simplistic structure makes for easy usability [2].

III. A Deeper Look into Model Checking

Model checking is the more popular approach in software verification. When it comes to information systems (IS), it concerns using an organized way to acquire facts and develop improved methods to derive the collection of evidence, measurements, and theories. In this context, Frappier and his associates implied that model checking had an advantage in IS over other formal validation methods since it has broader coverage and less human guidance necessary. Representative

of the main classes in model checking (explicit state, symbolic, bounded, and constraint satisfaction), they encompass a well-rounded look into the following six model checkers: SPIN, NuSMV, FDR2, CADP, ALLOY, and ProB [1].

A. Case Study: The Library

The case study presented is of a library system composed of entities (members and books) and associations (reservation, loan, renew, return). From this system, the relationship between the entities and associates generated properties that can be verified using model checkers. In consideration of IS, the properties of liveness and safety were distinguished. Liveness properties correspond to the system being alive and sufficient conditions enabling an action. Safety properties correspond to necessary conditions to enable an action. Note that the former case means the action is allowed but not forced, whereas in the latter case, it more so means what the user cannot do. Overall, within these properties outlined, the different model checkers were put into operation to view the effects of its individual characteristics in dealing verificationd [1].

B. Model Checkers and Specifications

1) SPIN: SPIN is "one [of] the first model checker[s] developed [that] introduced a classical approach for on-the-fly LTL model checking [1]. This means that a counter-example can be generated without constructing a global state transition graph. SPIN is an explicit state model that has its own input specification language called Promela, and for property specifications, it supports the temporal languages of LTL, CTL, and XTL. Promela can handle multiple processes simultaneously. Transitions may have to be computed and recomputed for each property verification, but state variables can be global and accessed by any process. Processes can communicate over a channel, synchronously or asynchronously. Due to the fact that SPIN checks finite state systems, it is also significant to note that this model checker uses propositional LTL. In regards to a Promela specification, a LTL formula can hold if it does so for every possible run - which can be infinite.

The ultimate application determined was for one process based on a producer-modifier-consumer pattern to describe the life cycle for each entity and association. From the majority of the properties, two patterns were derived: "'if action *A* can be executed then the state verifies *P* or 'if *P* holds then action *A* can be executed' where *P* is only true between *B* and *C* [1]. However, this pattern did not fit all of the properties for the library example. For those special cases, it was significant to not disregard these even if inexpressible.

2) NuSMV: NuSMV is a symbolic or bounded model checker, which denotes that it specifies finite state machines by representing the transition system as a Boolean formula; the latter form considering traces of a certain maximum length. A specification is compiled by module declarations, which in turn is composed of assignment constraints or transition constraints to model a system transition. An assignment constraint defines a value for a variable in the next step, and a transition constraint is a Boolean formula that restricts a potential next value. Each module can be instantiated by another since they are processed synchronously unless parameterized.

Implemented in the library case study, a systematic class structure was used. A class was encoded into a module to identify entities, and for each action a new module was created, which checked that a precondition is satisfied to then modify variables of entities. Consequently, all properties were expressed easily through CTL or LTL formulae.

3) FDR2: The model checker FDR2 is an explicit state model checker as well as an implicit state model checker. It uses the same language, CSP, for both model and property specification to make general verifications, and also builds the state-transition graph while compressing and checking properties concurrently. Under CSP, basic process algebra operators and data types are supported. Properties are checked using process refinement, of which there are three relations: trace refinement, stable-failure refinement, and stable-failure-divergence (or simply failure-divergence refinement). They are used to check safety properties; liveness (or reachability properties); livelocks, respectively.

Using a nominal/controller pattern, CSP was very useful to represent the IS entities life cycles. For properties, the appropriate relations of trace and stable-failure refinement came into play (failure-divergence refinement was not used in this study). Although the safety properties were easy to specify, reachability properties were a bit more complicated due to such cases as hidden association actions. For example, the hidden behavioral errors caused infinite internal action loops and made the states unstable. To override this error, Frappier and associates had to consider each association individually and disable events from other associations.

4) CADP: CADP is an explicit state model checker. The research by Frassier and associates focuses on LOTOS-NT for model specification, and XTL for property specification. The chosen specification is split into algebra operators for abstract data types, and a process expression. The algebra operators complement how assignment statements can be mixed with process expressions. Temporal logic properties are expressed by the aforementioned chosen property specification language. This takes a labeled transition system (LTS) input in the binary coded graph (BCG) format; the model specification is translated into LTS and then minimized into a trace equivalent so that the XTL properties can be checked against it.

There were realized issues right off the bat in utilizing this model checker. LOTOS-NT is similar to CSP and runs into the same problems that makes one have to commit hardcoding in accordance with the number of interleaves. Safety properties were defined using the following two patterns: an action *A* should not happen (1) between, (2) nor outside, two actions *B* and *C*. Liveness properties were written with ACTL and HML operators. Some properties did not have a correct formulation that could be derived.

5) ALLOY: Similar to NuSMV, ALLOY is a symbolic or bounded model checker. It employs first-order logic as its modeling (and property specification) language, and the only type

of terms are basic sets and relations, which are defined using "signatures." Constraints condition the values of signatures.

Three patterns were generated as a result of model checking with ALLOY: (1) when condition *C* holds, action *a* must be executable; (2) vice-versa condition from first pattern established; (3) *C* is a result of executing action *b(p)* that should enable action *a(p)*. These ideas are approximations, as the precondition must hold, the action executed, and the postcondition feasible. There are usually memory limitations, but with the fact that postconditions are normally feasible, the execution ability of an action is approximated on the precondition. Memory limitations could not be overcome when trying to express reachability, so it was resorted to describing how the trace in the transition system could be computed. Using ALLOY also brings about another difficulty when determining the valid library states where *C* holds. "These can be either characterized by a fact, which is error-prone to specify or by executing entity and association producers from the initial state of the system to automatically construct valid library states satisfying *C*, which is significantly less efficient to check" [1].

6) ProB: ProB is a constraint satisfaction model, and the research uses the B Method and B language. Organized into abstract machines, each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables [1]. Because it is acceptable to use simple set names for the sets, operations are defined using substitutions.

Similar in the case of SMV and ALLOY, there exists a difficulty of translating the ordering constraints in a precondition aspect and finding appropriate updates of state variables. Referring back to the patterns specified for the properties through SPIN model checking, these can also be applied in this ProB case. All properties are easily expressed in LTL and CTL. However, one of the properties (number 12) does not fit into the two patterns mentioned previously.

C. Analysis

To organize the analysis, Frappier and associates categorized their conclusions into looking at the model specification language, property specification language, execution time and number of entity instances, and tool support. Looking at the model specification language examination generally, all model checkers seem to sustain basic entity and association structure. There were exceptions in regards to parameters, and some model checkers were more suited in certain respects of representation at times, but overall support is provided. When investigating property specification language, there were specific model checkers that were obviously more efficient and reasonable to utilize. This section was inspected on a more deeper level of class for IS, and with a quick glance, ProB was the strongest form. However, in the case of execution time and number of entity instances, this model checker failed due to defects. The library case study had a minimum of three instances to check reservation queues greater than 1. The majority of the model checkers could not check more than three instances for each entity for at least one property. ALLOY was the most efficient for IS with an impressive 98 instances for almost all properties in under a minute. Simulators for each method were available and were straightforward, except for in NuSMV.

IV. Personal Application

I cannot say that I can directly apply this in my own professional activities, as I am currently not yet in a role that is technologically heavy, really. However, with my basic understanding, I can assume that a model checker, such as that detailed in this paper, may very well assist in software verification issues currently experienced. Let me also assume this in the place of someone who may actually be in such a position.

At present, the more traditional methods of debugging and testing are being used in the company, over allegedly more advanced methods such as theorem proving and model checking. Reports are run about two times a day, and these catch the more obvious ones running in the system. However, throughout work, employees also notice some setbacks or even customers notice some errors that hinder sales and progress. It is very obvious that this is a lax method of dealing with issues, as even though we can find bugs and resolve the small issues after discovery, no measures to be more proactive are taken. Tests are not even run regularly, and if conducted, are not necessarily run in full.

From the information that I have learned about model checking, it seems that this method would actually be much more efficient. Although it would possibly be more complex in the initial process of deriving the correct formulas, and defining the properties and attributes, the overall working process and stronger abilities in locating any system issues would be very useful and rewarding especially in the long-run. Looking at just the six models that were mentioned within the case study, I want to say that ALLOY, SPIN, or even NuSMV are the most appealing at this time. Utilizing ALLOY simply sounds appealing because the impressive number of (simple) properties it can process in a short amount of time. Otherwise, SPIN seems like a very classical model checker that has a usage case of simplicity and straightforwardness. Lastly, my choice of NuSMV is because in the case study, the properties could all be expressed easily through CTL or LTL formulae. I believe that this shows how adaptable this model is, and the class structure can apply well in the case of the specifications that the company can set.

V. Conclusion

We went from a more manual and responsive work of debugging and testing to a more proactive and preventive view of formal verification by theorem proofs and model checking. The research that Reinbacher, and Frappier and his associates completed shows how much model checking especially is growing in popularity. However, there are several different types of model checkers for different situations, and yet, findings show that there really is no one-size-fits-all. They have to be polyvalent, and should support both states and

events, for the best results and encompass all properties and specifications.

REFERENCES

[1] Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M. (2010, June 16). *Comparison of Model Checking Tools for Information Systems*. Universite de Sherbrooke

[2] Reinbacher, T. (2008). *Introduction to Embedded Software Verification*. Technikum Wien.