Formal Verification

An exploration into core concepts and ideas

Nathan Hillyer
Computer Science
Lewis University
Romeoville, Illinois

***Abstract*—This document constitutes an exploration into formal verification along with investigations into the current state of formal verification. Formal verification is using mathematical concepts in order to prove that a software or hardware system, or subset thereof, is bug free, or, as we will see, more accurately; it meets the specifications of the system designers. We will then evaluate an application of formal verifications, the AAMP5/AAMP-FV microprocessors. These processors had their microcode verified to a high degree and the result was nearly bug free once it went to production. We will also review two articles from the field and summarize their key points.**

Keywords—formal verification; software testing; hardware testing; pvs; aamp5; aamp-fv; proofs;

I. Introduction

Formal verification is the application of mathematics to the problem of hardware or software verification. By verification, we mean being able to prove that a system works in all conditions. Formal verification has historically been quite limited in its practical application, but the theory, in itself, of being able to prove bug-free software/hardware implementations, is an attractive proposition. Despite this, formal verification isn't very common. A couple things contribute to this state. Formal verifications requires a lot of computing power to be practical, and it has a high learning curve. This high learning curve was evident in the creation of the AAMP-FV processor, a descendant of the AAMP5 processor which used formal verification techniques in order to prove the processor was bug-free. The AAMP-FV costed almost an order of magnitude less then the AAMP5 to prove an equivalent number of instructions [1]. This suggests a high learning curve. The first major problem of the lack of computing power is close to being solves at this point thanks to Moore's law. This leaves us with the high learning curve as the current main barrier to entry.

II. Formal Verification

A. Speed of adoption

Formal verification lags behind our abilities to create complex computing information systems. This has primarily been due to the lack of sufficient computing power. It takes a certain amount of computing power to simply run a system, but testing the system under all possible inputs takes much more power than the system itself requires, so it's easy to see why formal verification would lag behind traditional software development. The other major problem with formal verification is the learning curve involved with learning how to use the tools for formal verification. These tools often come packaged in with their own programming language, making it hard to transfer skills between disparate tooling. This could be remedied by widespread adoption of a tooling-agnostic programming language which would effectively force major methods of formal verification to adopt. Another remedy would be advances in the user experience and design of formal verification tools.

III. Article Reviews

We will now evaluate two articles on the subject of formal verification: Formal methods: Practice and experience [2], and Survey of existing tools for formal verification [3].

A. Formal methods: Practice and experience

This article focuses on the practical experience of formal verification through the evaluation of surveys and case studies on the topic. It also presents its own survey and integrates this survey with the article's arguments.

The authors, Jim Woodcock et al., argue that an important first step in software engineering is to come up with the requirements of the project. They go on to share that formal methods are useful in creating a specification. This is backed up by a survey within the article: 93% of respondents said they use formal verification tools for the specification/modeling phase of software development. They define a specification as, "a technical contract between programmer and client to provide them both with a common understanding of the purpose of the software." Indeed, there are technical reports from NASA that covered the usage of formal methods to create specifications.

Model checking is a form of automated formal verification that checks a model or specification against various input in order to prove that it works as expected in all states. This has been a form of formal verification that has lagged behind because of the requisite computing power. The authors, however, indicate that usage of model checking has increased from 13% in the 1990s to over 50% in 2008. This may be attributed to the increase in computing power and the maturation of model checking tools.

The biggest question of formal techniques is how much value do they create in a traditional software development

setting where additional costs or time can make or break a project. Unfortunately, the author learned the most businesses don't keep accurate models for project costs. This may partly attributed to the nebulous nature of large scale development. Over 50% of respondents said that formal techniques either had no observable effect on cost or they didn't have enough data to determine. 7% of respondents said the it had a negative effect on cost, whereas 37% said it decreased costs. This is, perhaps, the more valuable metric. 5x as many people find it decreases cost than increases it. This could partly be attributed to participants' confirmation bias, but it appears, at the very least, that it can decrease costs in the type of projects evaluated within the survey. The authors readily admit that their 62 survey participants were selected with personal bias—there aren't enough people using formal techniques to facilitate a random sample. The projects within the article are all safety/ mission critical systems, and it's plausible that the survey results only loosely apply with day to day software development.

The authors also tried to answer an important question: Does it work? The resounding answer appears to be yes. 92% of survey respondents said formal techniques increased quality. There is also evidence in the case studies supporting an increase in quality. The Transputer project only found two bugs in the formally verified processor microcode. One was due to a bug in the program used to translate the formal specification into code and the other bug was due to hand optimization of some assembly code. The developers of the SACEM software, used for signaling and control in Paris commuter trains, claimed that their system was safer as a result of formal verification. The Mondex Smart Card project was determined safe enough to be the first product ever to obtain the ITSEC Level E6 certification. The AAMP project also found positive results. The act of coming up with the formal specification itself exposed two errors. The AAMP engineers also slipped in a couple subtle bugs with the intent of making them hard to detect. Despite this, the formal verifiers caught both bugs, suggesting that formal verification increases software quality. The Airbus project, along with major productivity improvements, found a significant reduction in programmer error when implementing a formally verified system. The Maeslant Kering Storm Surge Barrier project highly recommended formal verification methods and the developers felt that the code was of higher quality do to their adherence. The Tokeneer Secure Entry System project also reported positive results, and there was only two bugs found by independent testing. One was in the implementation code itself, and the other was due to limitations in the formal verification toolset. After the toolset was updated, it also caught the bug. The Mobile FeliCa IC Chip Firmware project found 278 bugs while using formal methods.

The most negative metric on the survey was how much time was spent on formal verification. 12% of respondents indicated a deleterious effect on their timelines, but the Transputer project found that the development of the floating-point unit with formal verification techniques was 3 months faster than the alternative informal development that ran concurrently. Of course, this begs the question as to how similar the conditions and resources were between the two projects, but it appears that the case study source was enthusiastic about the time saved in the project nonetheless.

The authors feel that the application of formal techniques is most useful in high-intebgry critical applications. They also observe that formal techniques are usually used by specialists and aren't simple enough for widespread adoption among the programming public. They feel that formal verification techniques deserve targeted research. They have noted a success in the area of lightweight tooling rather than fully formal techniques, and this may be the compromise we'll see for some time. Currently, the success is limited by the quality of tool support. Perhaps current initiatives, such as a repository of verified software, will spur increased quality in formal verification tooling.

B. Survey of Existing Tools for Formal Verification

This article surveys the landscape of existing tools for formal verification. It begins by introducing formal methods, then explores tools for abstract models, tools for checking hardware description languages, tools for checking correctness of software, and tools to create a provably correct design.

The authors remind the reader that formal methods largely fall into two categories: Model checkers and theorem provers. The primary difference between the two types of tools is how automated they are. Model checkers are generally more automated and theorem provers generally require a lot of human interaction. Theorem provers are more capable and flexible, but require more expertise to utilize.

The tools comparison begins with an evaluation of tools for checking abstract models. These abstract models are usually created in a domain specific language because practical languages aren't geared towards formal verification. The tools evaluated for Spin, Uppaal, SMV/NuSMV, FDR, Alloy, and Simulink Design Verifier. Spin is geared toward concurrent systems, and is one of the first model checkers. Uppaal is a model checker that includes the time as a variable. NuSMV, the successor to SMV, is designed for hardware design, but could be used in other use cases. FDR uses Communicating Sequential Processes, a language for describing computer systems, for the user to create a model of the system and a property to check. Alloy uses it's own language, called Alloy, for the creation of a model. It is usually used for analyzing the structure of data and cannot analyze time based properties. The Simulink Design Verifier is a tool that verifies a Simulink design. Simulink is a language that has signal processing and control logic built into it. There are surely more tools one could evaluate with in the abstract model space, but this briefly covers six of the more common tools.

The tools comparison continues by evaluating tools that check hardware description languages. The advantages of these tools is that they don't require a separate model from the design —as long as it is written in a supported hardware description language. Perhaps this practicality is a curse, because these tools are expensive. A yearly license can cost as much as $400,000 USD, and all the available tools are commercial. Mentor Graphics has a tool called Quest Formal as part of it's suite of various tools related to electronic design. It has 7 different methods of analysis. Solidify by Averant is another tool. It has 2 different methods of analysis and appears to have

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors)*.

fallen behind the times, as it is buggy and has incomplete support for the VHDL language. JasperGold is a formal verification software that has many different methods of analysis. It has the appearance of being the leading tool by the metrics of performance, feature set, and market adoption. Incisive by Cadence, is another tool for verification. Cadence has a large toolset for electronic design, and Incisive is one of such tools. These are five of the most prevalent tools in the hardware design language software verification space.

There is also a limited number of tools specifically made for the correctness of software. They generally limit the subset of features within a practical language to make it amenable to formal verification. Frama-C is one such tool that performs static analysis of C code. Unlike a traditional tool for static analysis, Frama-C gives the user the ability to create specifications and prove that the code matches the specification. BLAST is another tool aimed at checking correctness of C programs. It's a fairly advanced model checker that integrates theorem provers within the model checking process. Java Pathfinder is a model checker for Java byte code. Because it analyzes byte code, instead of Java itself, it has wide applicable to the proliferation of languages target the Java Virtual Machine. SPARK ADA is used as a subset of the ADA language and can also be used to check proofs. Malpas, MALvern Program Analysis Suite, is another tool in the software arena. It, like Frama-C, is primarily a static analyzer, but it can also verify specifications. These are a few of the tools available for software correctness.

The authors also evaluate a couple cools for creation of a provably correct design. The Vienna Development Method is one such tool that is used to model computer programs. Z, B, Event-B, and Rodin are another toolset. Z and B are used for specifying computer programs, and Event-B is used to crate a state machine representative of the system. Rodin is the tools that help design and analyze Event-B models. Rodin and Event-B have been used to verify correctness for mission critical software such as railway switches.

IV. THE AAMP FAMILY

AAMP are processors created by Rockwell Collins. The project for the AAMP5 processor was able to use the Prototype Verification System to formally verify the microcode of the processor. The results of the project were the participants realizing that it is technically possible to prove microcode correctness and that engineers can read and write formal specifications. They were able to catch two errors by creating the specification alone. The major downside of formal verification within this project was that the cost was high. It took approximately 300 hours per instruction given to the model checker. They tested how well the proofs of correctness performed in a real system by introducing two bugs designed to be difficult to detect. The company in charge of the verification caught both of them.

The AAMP-FV project was one that was undertaken after the AAMP5. They attempted to reuse as much as possible from the previous project and see how much time it saved. It turned out that the initially the AAMP-FV project costed an order of magnitude less to formally verify. However, over time more complex instructions required new techniques and began to experience delays and a steep learning curve for engineers involved in the previous project. This caused the participants to conclude that it's difficult to introduce new techniques, such as formal verification, within an industry that has widely accepted approaches within a well understood problem domain. They felt this way because there is not much information available to apply formal verification to a new setting. They also concluded that the PVS language and theorem prover were not as hard to learn as had been previously thought [6].

The AAMP case exhibited formal verification in a very large, mission-critical project. While there were some problems, namely in the ramping up of knowledge and learning curve, the project stakeholders felt that formal verification was a welcome and useful additional the process of developing a microprocessor.

REFERENCES

1. J. P. Bowen and V. Stavridou, "Formal methods and software safety," *Safety of computer control systems,* pp. 93-98, 1992.
2. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM computing surveys (CSUR),* vol. 41, no. 4, p. 19, 2009.
3. R. C. Armstrong, R. J. Punnoose, M. H. Wong, and J. R. Mayo, "Survey of existing tools for formal verification," Tech. rep., Sandia National Laboratories2014.
4. S. P. Miller and M. Srivas, "Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods," pp. 2-16: IEEE.
5. M. Srivas and S. P. Miller, "Formal verification of an avionics microprocessor," 1995.
6. S. P. Miller, D. A. Greve, M. M. Wilding, and M. Srivas, "Formal verification of the AAMP-FV microcode," 1999.