Panel: Given that Hardware Verification has been an Uphill Battle, What is the Future of Software Verification?

Sandeep K. Shukla
FERMAT Lab
Center for Embedded Systems for Critical Applications
Virginia Tech.
Blacksburg, VA 24060
shukla@vt.edu

Tevfik Bultan
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106, USA
bultan@cs.ucsb.edu

Constance Heitmeyer
Naval Research Laboratories
Washington, D.C, USA
heitmeyer@itd.nrl.navy.mil

Abstract

This industrial panel is organized to discuss the views, experiences and opinions of formal methods practitioners from design automation, hardware and software industries, in order to understand the industrial needs and trends in using formal methods. In particular, we discuss the current thrust on application of formal verification in software development, and what hardware formal verification experiences bring to bear for formal software verification.

1. Introduction

In the last decade, the efforts and successes in the industrialization of formal verification (FV) have mainly been in their application to hardware. Recently, there has been a growing interest in application of FV to software both in academia and industry. Interestingly, research in FV has started in the software arena, exemplified by Floyd's use of assertions, Hoare logic, predicate transformer based designs of Dijkstra and later on Manna and Pnueli's work on temporal logic, and Clarke and Emerson's work on synthesizing synchronization skeletons of concurrent programs. However, the research on software verification never seemed to mature enough for industry adoption. Uses of formal verification techniques in B, and Z-specification languages, OBJ or similar theorem proving based approaches, Larch theorem proving environment for algebraic specifications, application of symbolic execution to various programming languages, process algebraic approaches such as CSP, CCS etc, remained in the confines of academia. Software development for safety critical applications such as avionics, and command & control systems for defense is an exception and some limited applications of FV especially by European companies working in these areas have been reported. However widespread use of FV in software industry has not materialized in the past. In contrast, during 1990s, we have seen an explosion in interest in FV by hardware design companies, such as IBM, Intel and Motorola. This interest peaked immediately after Pentium Floating Point division bug, which cost Intel almost half a billion dollars. Hardware verification in the form of model checking seems to be the most well known FV application in the industry since then. Recently, we observe that companies such as Microsoft and government agencies such as NASA are showing a growing interest in application of FV to software development. Also formal modeling approaches such as State Charts, SCR and other specification languages and frameworks seem to be getting a foot hold in software industry.

The questions we ask of this panel in this context is whether there are hopes in providing reliability and correctness guarantees in software through formal verification when doing so in hardware case has been quite difficult. The inherent premise of this discussion of course is that hardware verification seems easier than software verification, especially due to the structured nature of hardware description languages compared to high level programming languages. However, there also, that premise is getting weaker since the distinction between hardware and software is get-

ting blurred in this age of embedded ubiquitous computing, where hardware/software co-design seems to be the theme of design methodologies.

2. Hardware vs. Software?

In the current age of ubiquitous, pervasive embedded systems, hardware and software designs are often done together, in the form of co-design. Often times, what was implemented as software in previous designs, is now being implemented in hardware, and vice versa, based on the performance requirements and other factors such as cost, power consumption, and time to market constraints. Interestingly, software itself has many different levels of complexity, structure, and intrinsic qualities. For example, real-time embedded software systems used in avionics control, or command & control systems are akin to hardware in the sense of their structural nature, strong requirements in terms of performance and response times, and the criticality of correct functioning. On the other hand business software is often quite different and very abstract in its data structures and functionalities, and built on multiple layers of software stack. Device drivers on the other hands are again quite structured in the sense that they are often interrupt driven, interactive against real hardware, and have strong correctness and performance requirements.

Researchers and developers familiar with mission critical software design and development report that formal methods have proven useful in making correctness guarantees. On the other hand, developers of business software, or networking software are not known to use formal methods. So there are apparently certain niche areas of software design that could make use of formal methods, and learn about formal methodologies and techniques used in hardware design context.

3. Topics of Discussion

We ask the panel the following questions:

1. What is fundamentally different between software and hardware formal verification? Is software formal verification inherently more complex than hardware verification?

2. As we know, semiconductor industry still hurts from the verification bottleneck. 70% of the project time is spent in verification, according to some figures. Formal verification, back in the early 90s promised to decrease this time. Intel Pentium 4 had a team of 20 or so formal verification engineers, and in CAV 97, the manager of that team claimed 100% chip verification. However, we know that no chip has had more than 25% of its logic formally verified, if not less. What are the reasons for the gap between the hopes then, and reality now?

3. Is formal verification of software is in the same stage as was hardware verification around 1994 (a lot of promise, lot of activities in academia and industry, a lot of hopes from management)? Is it time for researchers to shift focus from hardware verification to software verification while there are still challenges remaining in hardware verification?

4. In the hardware industry there has been a standardization effort for assertion languages which can be used both for FV as well as dynamic verification. Should there be some effort like that in software verification?

5. Culturally, universities do not teach verification well. Most university curriculum has many many design courses, some testing courses, and the FV courses are more based in schools with FV research groups but they focus more on how to build FV tools and not how to use FV effectively on designs. In software, the culture is even worse and the coverage of formal methods in US schools is minimal. In order for FV to be used in software development, should this situation need to change? How can we affect the changes there?

6. Software verification has a long history going back to sixties. Why hasn't it been successful in the past and what is different now that there are some of us who believe that it will be successful in the future. (It is hard to define success but one possible definition would be wide use of software verification tools by the software developers.)

7. Will software verification have its impact at the programming stage by developing tools tied to programing languages (such as C, Java) or at the design stage by developing tools tied to design languages (such as UML)? A related question is will there ever be widespread use of formal design languages in software? An extension to this question is what is the interplay between languages and verification techniques? Should the goal be developing automated verification tools for existing languages, or developing new languages which exploit the verification technology?

8. The foundations of static analysis techniques used in compilers and automated verification techniques such as model checking are closely related. Hence, one can look at software verification as an extension of static analysis. So the questions is what is more likely: Software verification being part of advanced compilers generating more and more sophisticated error mes-

sages or software verification resulting in verification tools independent of compilers.