Haifa verification conference 2007

Karen Yorav

Published online: 3 September 2009
© Springer-Verlag 2009

Abstract This special section contains a selection of contributions originally presented at the Third Haifa Verification Conference (HVC'07). The scope of this conference covers all types of verification of both hardware and software systems. While there is widespread agreement on the importance of verification, it is clear that different systems require different approaches. Several distinct fields of research have developed, devoted to either software or hardware, or to a particular verification approach such as formal or testing/simulation. Each of these paradigms has an extensive publication history and its own dedicated conference. Yet there is much to be gained from sharing knowledge and experience. HVC's goal is to serve as a venue for researchers from all fields of verification, enabling them to exchange ideas and learn from one another. It is our hope that by gathering these experts together in one conference, we are fostering the emergence of new trends that combine ideas and insights from different domains.

1 Verification paradigms

Verification approaches are generally categorized as dynamic or static. *Dynamic verification* means the system is executed, probably numerous times, and faults are identified by examining the progress and the results of each run. *Static verification*, on the other hand, attempts to verify a system without running it, by analyzing the system description in some way, without executing it on specific inputs. Of course, there are many interesting methods that are hard to categorize, but typically software testing and hardware simulation are considered dynamic

approaches, while formal verification and static analysis algorithms are considered static approaches.

For the sake of simplicity, we focus on two types of systems: software and hardware. Many similarities exist between the two systems. Both use variables (or signals) that have types and are assigned values that may change over time. They both receive inputs, perform calculations, and output the result of the computation. Yet there is an inherent difference between the two, which seems to be in the proportions. While in hardware it is given that the values of all signals are updated in parallel at each clock cycle, in software the amount of parallelism is much smaller, with significant amounts of computation being executed sequentially. In hardware the most critical parts are often Boolean, while in software the focus is on variables with large types. Admittedly, these are all gross generalizations and for any of these generalizations it is easy to come up with a contradicting example. Yet these differences led researchers to diverge into separate communities. The cross product of the two categorizations, dynamic vs. static and hardware vs. software, leads to four basic domains of research in verification: software testing, hardware simulation, formal verification of software, and formal verification of hardware. This is depicted in Fig. 1.

Software testing is a dynamic technique based on repeated execution of a program with different inputs. Information about the execution is gathered and analyzed. Often the program is “instrumented” by adding code in between original program commands. The instrumentation code is responsible for bookkeeping whatever information is needed for verification. The main advantage of this method is its scalability; if a program can be executed, it can be tested. The main drawback of this method is that it is not exhaustive, so there is always the concern of missing bugs. This is why one of the main issues with testing is *coverage*, which measures how extensive the verification is. Coverage is measured by

K. Yorav (✉)
IBM Haifa Research Laboratory, Haifa, Israel
e-mail: yorav@il.ibm.com

	Hardware	Software
Dynamic	Hardware simulation	Software testing
Static	Hardware FV	Software FV Static analysis

Fig. 1 Verification paradigms

defining a set of events (a coverage metric) such as branches being taken or not, commands being executed or not, etc. During testing, coverage statistics are gathered and progress is measured by the percentage of events that were encountered. Software testing is the main means of verifying real-life software systems.

Hardware simulation¹ is also a dynamic technique, based on repeated simulation of the hardware design. In fact, the characteristics of software testing described above are true for hardware simulation as well: its main advantage is scalability, its main drawback is lack of exhaustiveness, and coverage is used to measure progress. However, there are some notable differences. Hardware can not be executed directly; it needs to be simulated. The types of events defined for coverage are very different. Also, the simulation model of a hardware design is basically deterministic, which makes it easy to re-produce a faulty simulation run, while software systems (in particular parallel programs) are tested in their real environment, where they interact with an environment other than the test input (such as a file system or an operating system) and are dependent on scheduling mechanisms. This causes difficulties in re-producing bad tests. Another difference is that in software it is easier to separate the control from the data than in hardware, which leads to different types of abstractions being used for software and hardware systems. Finally, perhaps the most intriguing difference is the expectation of quality. In hardware designs, any fault that is exposed to customers can generate significant loss of income since the users expect flawless execution. Thus, the amount of effort invested in simulation is huge, both in manpower and in computational effort. In software, however, it is considered reasonable to release a product with several, or even many, known faults, and the cost of fixing a fault at the customer's site is relatively low. As a result, software developers are not inclined to invest more time and energy in testing than is necessary.

Formal verification techniques are based on using mathematical tools to prove that a property holds on a particular system, or else to produce a counterexample. The focus of formal verification is on exhaustiveness, at the expense of scalability. Formal verification techniques tend to suffer from the *state explosion* problem, which limits their applicability to smaller components. Naturally, overcoming this limitation

is the main focus of research in the field, and great progress has been made in the last two decades. While most formal verification methods can be applied to both hardware and software, there is a big difference in capacity. Explicit-state verification—where the FSM describing all possible behaviors of the system is explored state-by-state—has been used successfully for software but is not as useful for hardware. On the other hand, symbolic reachability analysis has made great progress in the verification of hardware systems but has not shown promise for software.

Finally, static analysis algorithms are used to analyze the behavior of a program without running it. These algorithms are based on the program code and its structure and can prove that some things can or cannot happen. These algorithms are highly language-dependent and hence do not easily migrate from software to hardware.

2 Cross-fertilization between domains

Cross-fertilization between the four disciplines described above can manifest itself in two ways. In some cases, an idea that was successful in one domain is migrated to another domain, while in other cases successful ideas from two domains are combined together to produce a new approach. A thorough literature review of all such examples is clearly beyond the scope of this brief exposition, so instead we focus on a few specific examples. We observe that given the matrix from Fig. 1, ideas tend to migrate across columns and are combined across rows. We see this in the examples below.

2.1 Migration of ideas

Coverage [5] is probably the most notable example of an idea migrating from software to hardware. Emerging in the early sixties, the notion of using coverage metrics for estimating the quality of testing was originally developed for software, and has migrated to the world of hardware simulation. Today, when the bulk of hardware design verification work is done using simulation, coverage is extremely important.

SAT-based bounded model checking [1] is an example of an idea migrating in the other direction, from hardware to software. Originally introduced in 1999, the method is based on reducing the problem of model checking up to a specific bound into a satisfiability problem. Given a hardware design and a bound k , the method produces a propositional formula that encodes a faulty computation of the design that is k cycles long. This formula is satisfiable if and only if there exists a counterexample of length k . The formula is given to a SAT-solver, which in turn either finds a satisfying assignment or proves that there does not exist such an assignment, thus proving that there are no bugs within the first k cycles of all computations. The success of this technique for

¹ The term “testing” in hardware refers to the verification of a chip after manufacturing. The hardware analogue of “software testing” is “hardware simulation”.

hardware designs spurred a wave of research into SAT-based algorithms for model checking. Shortly after, similar ideas were applied to software in the form of tools that translate a software verification condition into a propositional formula such that a satisfying assignment corresponds to a counter-example.

Another technique that migrated along the same route is automatic test generation. Simulation or testing are and probably always will be the main vehicle for verifying systems. Random stimuli easily traverse the most common behaviors of the system. The challenge is to drive the system into corner cases and rare occurrences, which is where the tricky bugs lurk. The idea behind automatic test generation is to automatically generate inputs that will drive the system under test into uncovered corners. This can be done in several ways: using constraint solvers, based on an abstract model of the system, or by invoking a model checker. These techniques are well established in hardware, and are making their way into software as well.

2.2 Merging of ideas

Perhaps more interesting than the mere migration of ideas is when two different methods are efficiently combined. One popular way to do this is by harnessing the analytical powers of formal verification technology to make dynamic verification more effective. An example of this situation is *target enlargement* [7]. This is a technique used for hardware simulation, in which the set of target states (or a coverage goal) is enlarged using backwards reachability, making it easier to hit. Usually, because of the size problem, the formal algorithm works on an abstract model of the design. On this model, it is possible to perform backward image computation from a target state for a certain number of steps. The result is an over-approximation of the set of states from which the target can be hit within k clock cycles. The original target function is replaced by a new target function that represents this larger set of states, resulting in a target that is easier to hit.

Another example, this time from software testing, is the DART tool [2], which uses formal verification technology to find inputs that drive testing to uncovered code. Random inputs are used to run the program, causing the traversal of a specific path in the control-flow graph of the program. During the run, DART computes a symbolic expression that describes input constraints to be used for the next run. These constraints ensure that the next run traverses a different control-flow path—taking the else-branch instead of the then branch of an if-statement, or going through a loop one more time. This enables DART to perform a systematic search of the set of paths in the control-flow graph. The computation of constraints is based on symbolic execution technology, which in general cannot handle life-size programs, but is used here only on the conditions in if-statements. Thus this

method elegantly combines the strength of a formal verification technique with the scalability of dynamic verification.

Similarly, simulation can be exploited to improve formal verification techniques. For example, BDD-sweeping [4] and SAT-sweeping [3] are reduction algorithms in which potentially equivalent signals in the design are identified, proven to be equivalent, and then merged. The bottleneck of this technique is the process of checking the equivalence of every pair of signals. To limit the number of checks performed, the algorithm starts with a (gross) over-approximation of an equivalence relation among signals, and then uses simulation to refine this relation before it is checked. Simulation is an excellent tool for this refinement process because it is fast and able to cover a large part of the state space in very little time, yielding a relatively good approximation.

The examples above show that merging complementary techniques can be both powerful and useful. For such ideas to be conceived, it is necessary for experts in different fields to meet, discuss the challenges they are facing, and exchange knowledge and experiences. The Haifa Verification Conference was conceived with this goal in mind, and with each year we see more interest in the conference and the topics presented there.

3 The papers

The papers chosen for this special section represent the top papers presented at the 2007 Haifa Verification Conference, and also an invited paper by the winners of the 2007 HVC Award—Corina Păsăreanu and Willem Visser. This collection gives a taste of all the flavors of verification—formal, dynamic, software, and hardware.

GenUTest: A Unit Test and Mock Aspect Generation Tool by Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai.

This work won its authors the best paper award at HVC'07. The paper presents a method for generating unit tests from system tests. Automatic test case generation is interesting and important for dynamic verification of both hardware and software. While this paper uses such notions as aspects and mock objects, which are software specific, one cannot help but wonder whether the core ideas can be transferred to work for hardware designs. After all, the dynamic verification of hardware designs is most often executed on a software model. It would be interesting to learn whether similar ideas could be used to automatically generate a testing environment for an internal block or unit, based on the simulation execution of the whole chip.

Using Virtual Coverage to Hit Hard-to-Reach Events by Laurent Fournier and Avi Ziv.

This paper comes from the world of simulation-based verification of hardware designs. It presents a novel method

of steering the simulation to hit rare events. The use of a learning algorithm to cover goals requires some kind of structure in the coverage space, which the learning algorithm discovers. For some types of goals this structure does not exist. This paper proposes artificially adding virtual coverage goals to generate such a structure and facilitate better learning. While these ideas were conceived in the context of hardware simulation, it is quite possible that they can be applied to software coverage as well.

Dynamic Testing via Automata Learning by Harald Raffelt, Bernhard Steffen, and Tiziana Margaria.

Learning algorithms have become a powerful tool used in many areas of computer science, and in verification in particular. They help make sense of intricate behaviors and enable the creation of system models without knowledge of the inner workings. In this paper, an automata learning algorithm is used to create a model of the system being tested, and this model is used to drive further testing.

Exploiting Shared Structure in Software Verification Conditions by Domagoj Babić and Alan J. Hu.

Even though just a few years ago formal verification of software was considered a distant dream, it has now become a very active and successful research topic. This paper is a good representative of this new trend with an optimization for proving verification conditions, driving forward the capabilities of the technology. Here the authors consider that typically many verification conditions share sub-expressions, and find a way to exploit this to speed up the proof for subsequent verification conditions.

Symbolic Execution and Model Checking for Testing by Corina Păsăreanu and Willem Visser.

Last but not least, this paper is by the winners of the 2007 HVC Award. The paper summarizes a body of work that combines symbolic execution and model checking. In choosing this paper as the winner of the first-ever HVC Award, the

award committee recognized this research as both a theoretical step forward and a practically useful study.

Acknowledgments Several of the examples given here were taken (with permission!) from Alan Hu's excellent talk at HVC'07, where he discussed the merging of ideas between simulation and formal verification. An extended abstract of that talk can be found in the proceedings [8]. I wish to thank Alan for inspiring conversations. Other examples were taken from a paper by Shmuel Ur and Avi Ziv [6], where they discuss the similarities between software testing and hardware simulation, and the need for cross-fertilization between researchers in these domains.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) (1999)
2. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI) (2005)
3. Kuehlmann, A.: Dynamic transition relation simplification for bounded property checking. In: Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design (2004)
4. Kuehlmann, A., Krohm, F.: Equivalence checking using cuts and heaps. In: Design Automation Conference (1997)
5. Marick, B.: The craft of software testing: subsystem testing including object-based and object-oriented testing. Prentice-Hall, Inc., Upper Saddle River (1995)
6. Ur, S., Ziv, A.: Cross-fertilization between hardware verification and software testing. In: Software Engineering and Applications Conference (SEA'02) (2002)
7. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: Design Automation Conference (1998)
8. Yorav, K. (ed.): Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23–25, 2007, Proceedings, volume 4899 of Lecture Notes in Computer Science. Springer, Berlin (2008)