Cross-Fertilization between Hardware Verification and Software Testing

Shmuel Ur Avi Ziv
IBM Research Laboratory in Haifa
Haifa University
Haifa, 31905
Israel
email:{ur, aziv}@il.ibm.com

ABSTRACT
For many, hardware design verification and software testing may seem like separate disciplines. Yet, significant similarities exist between software development and hardware design, and the successful adaptation of techniques originally developed for one field for use in the other, suggests that these disciplines are related. One prominent example is code coverage, first developed for software testing and now commonly used in hardware verification. Another example is the FSM based test generator, developed for the verification of hardware modules and now successfully employed for software testing. Moreover, some techniques, such as reliability estimation, were developed for hardware, changed and adopted for software, and now again show their usefulness with hardware. This paper analyzes the similarities and differences between hardware verification and software testing. It provides a short survey of methodologies and techniques that were developed for one field and later adapted for the other. We then speculate, based on experience in both fields, which hardware verification technologies can potentially impact software testing.

KEY WORDS
Software testing, Hardware verification, Software engineering

1 Introduction

Testing is one of the costliest stages in the industrial software development cycle. Testing usually constitutes between 40-80% of the development process (e.g., 70% for Microsoft) as compared with less than 20% for the coding itself [1]. Software testing, therefore, gets a lot of attention in the software engineering community, and many new technologies and methodologies, such as software static analysis techniques [2] and orthogonal defect classification [3], are becoming common practice.

The hardware industry faces similar issues. The increasing complexity of current designs and the huge cost of delivering a faulty product [4] have led to a growing investment in functional verification and in the development of new technologies and methodologies in this area. This include sophisticated random test program generators [5] and model checking tools [6].

The design and implementation of software systems have strong similarities to the logic design and implementation of hardware systems. In both, the process starts with a high level design of the entire system, which is then broken into smaller units. Each unit is implemented and tested by itself. Next, the whole system is constructed by combining the units. In addition, the languages used for the development of hardware systems, like VHDL [7], resemble software programming languages. We would therefore expect similar technologies to be used, at least for software where the cost of faults is high; however this is not the case.

Testing of software systems and the logic verification[1] of hardware designs also follow a very similar process. Input patterns are generated and the actual behavior of the module is checked against its expected behavior, as defined by the specification and architecture.

Although the software testing and hardware verification communities share the same goals and face similar issues and challenges, they use different philosophies in their approach. The main reason for the development of the two philosophies is the limited interaction between the two communities. They hold separate conferences, and tool vendors work only in one domain. Even in organizations where the two communities share the same premises, they have little awareness of each other and rarely share their problems or solutions. This lack of interaction, or segregation, is more evident among users (the people who actually do the testing), but also exists among tool and methodology developers. One of the outcomes of this separation between the communities is that useful technologies that are developed, tested and proven to be successful in one domain, are rarely used in the other. Other reasons for the different approaches in the two communities include the similarities in projects done by hardware teams as opposed to the small resemblance between various software projects, the huge

[1]The two communities use different terminology. The software community uses the term ‘verification’ for formal verification, while the hardware community uses ‘verification‘ for the functional comparison between the specification and the implementation. The term ‘testing‘ is used for manufacturing testing. In this paper, we use the native terms of the two communities; that is, software testing and hardware verification.

cost of bugs in hardware, and the superior programming skills of hardware verification engineers.

Our experience shows that close interaction between the software testing and hardware verification communities can benefit both communities. We describe several techniques that were developed by one community and later successfully adopted by the other, including code [8] and functional [9] coverage, formal verification [10], and reliability estimation techniques [11]. We also discuss some emerging technologies in the hardware verification world that may be suitable for software testing, how they can be used, and what adaptations are needed in order to make them effective there. Specifically, we address two areas, testing of parallel and distributed systems, and the use of random test generators and reference models in software testing.

This, of course, does not mean that all the problems of one community can be solved by the other. There are many inherent complexities in software and hardware systems that require unique solutions that cannot be shared. For example, the software testing community developed many useful techniques for GUI testing [12], while the hardware verification community is dealing with techniques that check if two levels of description are equivalent [13].

Still, the common issues and challenges faced by the hardware verification and software testing communities, and the great benefits to be gained from sharing technologies and methodologies, lead us to the believe that intensive interaction and cooperation through common conferences, common working groups, and other venues is needed.

We use the Verification Technologies department at the IBM Haifa Research Laboratory as evidence for the benefits of close interactions between software testing and hardware verification. This department comprises groups that develop tools and methodologies for both software testing and hardware verification. The close contact and constant interaction among the people in the department allows us to closely examine the needs of each community and compare them with what the other community has to offer. This enables a successful transfer of technologies between the groups. As an example for such a successful transfer, we describe a model-based test generator called Gotcha-TCBeans [14], which was originally developed for hardware testing, and later adopted for software testing. This tool is now being used in several IBM software development laboratories.

The rest of the paper is organized as follows: In Section 2, we briefly describe the current methodologies and common practices in software and hardware development and testing, show the similarities in the design processes, and discuss the differences in testing methodologies. In Section 3, we describe several cases of technology and methodology transfer between the two communities, and discuss some hardware verification technologies that can be useful in software testing. In Section 4, we discuss the experience of technology transfer at IBM. Section 5 concludes the paper.

2 Current Methodologies in Hardware Verification and Software Testing

Software and hardware systems are both developed based on similar concepts and have much in common. These similarities are visible in most phases of the development cycle, from design through implementation, and on to the testing of the system. While the concepts underlying the development of hardware and software systems are similar, the methodologies and techniques used in their implementation are different. In many areas, the hardware industry, although much older than the software industry, lags behind. For example, high-level languages were used for many years in the software industry, before becoming common practice in hardware design in the 1990s.

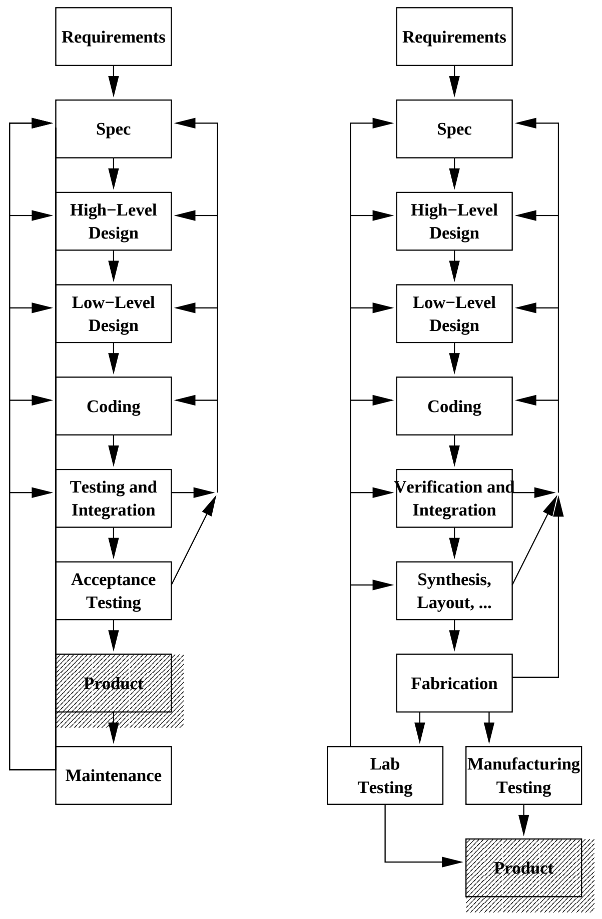
Testing is one area in which the hardware industry has an edge over the software industry. The main reasons for this phenomenon, are the reliability requirements of hardware systems, the high cost of fixing bugs in silicon, the higher cost of simulation in hardware as compared to program execution in software, and the highly parallel nature of hardware systems. Even though, in many cases, the quality requirements for software systems are lower, and less resources are used for testing, many techniques used in hardware verification, are, or can be, useful in software testing. Similarly, techniques developed for software testing are, or can be, useful in hardware verification.

In this section, we compare the processes used in the development of hardware and software systems. We start with a general description of the design and implementation phases of the process, and highlight the commonalities between them. There are development phases that have no equivalent in the other discipline. These are mentioned here for completeness but are beyond the scope of this paper. We also describe common practices used in hardware verification and software testing, compare them, and discuss the differences.

2.1 Design and Implementation Processes

Figure 1 presents a flow chart of the system development process for software and hardware. For both, the process starts with the customer specifying the requirements of the system, both functional and non-functional. The requirements are used to create the specification of the system, upon which the rest of the development process is based. In hardware design, this phase is taken very seriously, mainly due to the importance of non-functional requirements. In software, this step is sometimes partial and in many cases even non-existent [15].

The next step is the high-level design of the system; here, the system architecture, its main components and the interfaces between the components are defined. The high-level design is followed by the low-level design, where each component is broken into smaller units. These units are implemented during the coding phase. In hardware design, coding is usually done in hardware description lan-



(a) Software (b) Hardware

Figure 1. Design flow of software and hardware systems

guages, like VHDL [7], whose expression capabilities are similar to those of software programming languages. The high-level design, low-level design, and coding phases of the development process are usually accompanied by manual inspection of the results, i.e., the design documents and the code itself, in order to detect missing requirements, ambiguities, and conflicts.

When the coding phase is completed, testing (or functional verification) and integration of the system begin. Testing starts with the basic units. When these are integrated into components, testing is done on the components. This culminates in the integration and testing of the complete system.

For software systems, this is the end of the development cycle. However, before a system can become a product, it must go through acceptance testing, which uses a separate set of tests, often performed by the customer. After the product is released, the system enters the maintenance phase, which consists of fixing bugs detected in the field and developing new versions. This phase, particularly the addition of new features to the system, may introduce changes in all the previous phases of the design and implementation process.

In hardware, after the logic design and verification of the system are completed, several additional steps are required to convert the functional design into a chip [16]. These steps translate the functional description of the system into gates and transistors, and the necessary data for manufacturing the chip. Today, these translations are usually done with automatic tools [16]. Many of the non-functional requirements of the system, such as timing, power consumption, and area, can be checked only during, or after, the translations. Problems discovered in this stage can, in some cases, be fixed at the gate and transistor level. Therefore, equivalence-checking [13] tools are used to verify that the outcome of the translation is equivalent to the functional description.

Finally, when the layout is completed, the design is sent to fabrication and the actual chip is manufactured. This is not the end of the road for the chip. First the manufactured chip must pass a series of manufacturing tests to ensure that the manufacturing process worked properly. More importantly, the chip is brought back to the laboratory, where it undergoes extensive tests to detect functional and non-functional bugs, that escaped the verification process. In many cases, the correction of these bugs affects all the preceding stages, and additional fabrications are required before the chip can be released as a product.

2.2 Common Testing Practices

We base our evaluation on a number of survey papers [15, 17], on software testing practices, on internal evaluations at IBM, on surveys carried out by IBM and reports given at high-end testing conferences, and on our conversations with many tool vendors, testers, and verification people. The main differences between software testing and hardware verification are summarized in Table 1.

Software testing practices are by no mean homogeneous. However with a few exceptions, software testing, even in the more advanced companies, is a labor intensive process. It is hard to say if this is the cause, or the result, of the status of software testers. Compared to developers, testers tend to be less qualified, and to have inferior or non-existent programming skills. Many of the more qualified testers see testing as a stepping stone in the path to development. As a result, it is very hard to employ advanced tools and practices in software testing. Nevertheless, the software industry is very large and some of its segments have very different standards of product quality and required testing. For example, in aerospace and military software a 100% statement coverage is required.

There are a number of reasons why hardware verification usually results in a higher quality product than software testing. The most important reason is the stable testing environment (human and tools). The average time for processor development is four years, and usually the same team develops more than one processor. As a result, the people in processor verification can use many complex tools, many of them home made, that have been acquired

	Software Testing	Hardware Verification
Skill level of people	Low	High
Creation of tests	Manual	Automatic
Inspection of tests	Manual	Automatic
Tools used	Coverage Capture and replay Bug tracking (few)	Simulators Behaviorals Test generators, Coverage (few)
In house testing tools	Few	Many
Quality requirements	Vary from low to high	Very high, bug is a disaster

Table 1. Comparison between software testing and hardware verification

over the years. Another reason is that quality requirements are very high and are recognized by all including management. It is simply unacceptable for the customers to find bugs. In contrast commercial software is often released with a substantial number of bugs.

The main technique in functional verification is verification by simulation. The verification environment contains test generators, behavioral simulators, and design simulators. Tests are created by the test generators, and executed on the design simulator. The behavior of the design and actual results of the simulation are compared to the expected behavior and results generated by the behavior simulators. This process is fully automatic and is repeated many times. Coverage is used to monitor the quality of the executed tests.

Model Checking [10], a technology that is computationally intensive, has recently became viable due to the dramatic reduction in the cost of CPU time. *Reasoning bug finder service* [18], a computationally intensive software testing technology, is another technology following the trend of making the computer work very hard to find bugs. Tools which are CPU intensive, are bound to become more common in testing and verification.

3 Technology and Methodology Transfer

Although there are large differences in the approaches used in software testing and hardware verification, there are many examples of techniques and methodologies that were developed for one domain and later adopted successfully in the other. In fact, some techniques completed a full circle. They started in one domain, were later adopted and perfected in the other, and then made their way back to the original domain. In this section, we describe several such techniques and discuss the characteristics that contribute to their successful transfer. We also discuss several fields in which cooperation between the software testing community and the hardware verification community has the potential to yield great benefits in the near future.

3.1 Coverage

Coverage is probably the best example of a methodology that was transferred between hardware and software. Simply stated, the main idea of coverage analysis is to create, in a systematic fashion, a large and comprehensive list of tasks, and verify that each task was covered in the testing phase [8]. Coverage can help monitor the quality of testing, and locate weak areas in the design and implementation of a test plan. Generally speaking, coverage models can be divided into two groups: code coverage and functional coverage. Code coverage models focus on the code of the implementation, while functional coverage models deal with its functionality [9].

Code coverage is the oldest testing methodology used in both hardware and software. In code coverage tools, a task is created automatically for each program artifact (user choice of block, branch, method) and the user tries to cover all these tasks. Code coverage tools for hardware [19] and software [20] were developed and used at IBM Poughkeepsie in the sixties. Later, the use of code coverage techniques and tools became common practice in the software industry and many code coverage models were developed [8]. In recent years, with the growing use of hardware description languages, code coverage tools for hardware languages are becoming more popular and have led to the creation of new coverage models, such as state-machine coverage [21].

The easy transition of code coverage techniques from hardware to software and back to hardware was enabled by two main factors. First, as both hardware and software are written in high level languages with similar expressiveness, coverage of the language is done in virtually the same way. Also, code coverage tools are easy to use and can be easily integrated into an existing testing environment.

Functional coverage is the practice of creating application-specific coverage models and measuring the coverage of the tests on these models. An example of an application-specific coverage model is verifying for a transaction system that every possible transaction was executed and received every possible response. This technique has been used in hardware verification in the form of application-specific coverage tools such as Covet [22].

As discussed above, the stability of the hardware verification environment in terms of personnel and application types has fostered many home-made tools including functional coverage tools. In 1995 an application-independent functional coverage tool — Comet [23] — was created with which the user can define application-specific coverage models and measure them. This tool increased the use of functional coverage, since it eliminated the need to write a new tool for each coverage model. The tool is now used at all IBM processor development laboratories. In software, even though a simple-to-use, functional coverage tool named Focus [24] is available, it is still not used. The reason is that the implementation of functional coverage requires significant expertise, and the benefits offered in terms of increased quality are often beyond the needs of software systems.

3.2 Code and Specification Inspection

Code and specification inspection [25] is another example of a technique used in both software testing and hardware verification. Inspection is the practice of reviewing, usually in a group with predefined roles for each member, all the lines of code that are written. Inspection is an extremely effective technique for bug discovery, mainly at the unit level. In many cases, bugs found by inspection are very hard to find in simulation. The technique of code inspection is similar in both hardware and software, even though different inspection checklists are used. Hardware, which tends to have much better specs, also use spec inspection extensively before the code is written. This is acknowledged as a good practice in software but is generally not adhered to.

3.3 Formal Methods

Formal methods are mathematically based languages and techniques for specification and verification of complex software and hardware systems. The formal method that is most commonly used today is model checking [10]. The idea behind model checking is to specify a set of (typically temporal) properties for checking, and then examine every possible state of the system and verify that these properties hold. The main drawback in the use of model checking is the vast number of states that have to be examined and the complexity of the checked properties. Advances in model checking technology that improve the data structures used in symbolic model checking [6] and increase computing power have enabled model checking tools to handle more serious problems. Today model checking is a technology which the hardware industry not only uses, but also invests in heavily.

Model checking is ideally suited for hardware verification, since hardware design is usually defined as communicating finite-state machines (FSMs). In software, on the other hand, it is much harder to extract small state machines needed for model checking. Therefore, in software, model checking as well as other formal methods such as theorem proving [26], are used mainly in the verification of specifications (when the specifications are formally written). While there is a lot of theoretical work, very little is actually applied in the industry. The notable exception is the use of model checking in the testing or verification of protocols, especially in communication. Communication protocols are actually very similar to hardware control, since both are written as state machines. We believe there is great promise in formal methods for software testing; it has become a common tool in hardware verification and has not yet made inroads into software testing.

3.4 Defect Analysis

One of the more important roles of the testing process is to assess bugs, their causes, and their impact on the tested system [11]. This assessment can pinpoint weak links in the development and testing processes, and provide vital predictions of the effort needed to reach the desired reliability [27].

Analytical modeling is the main method used for the prediction of software reliability and estimation of the number of bugs left in the system. The parameters of the model (or models) are estimated from the available history of the failure data. The model is then used to predict relevant measures, such as reliability, bug discovery rate, etc. [27].

The roots of this technique are in statistical models that were developed to assess the reliability and expected life-time of mechanical and electrical components. The technique was adopted by the software reliability community, and many models that fit the behavior of a software system under testing were developed [11]. Today, the use of this technique is widespread; several tools that help manage bug discovery data and provide the needed predictions are available.

More advanced management techniques, such as ODC (Orthogonal Defect Classification) [3], add root cause analysis techniques (i.e., methods that analyze the cause of bugs) to statistical modeling, and assist in pointing out weak links in the development and testing processes. ODC provides a set of attributes that help quickly capture the semantics of each defect. Statistical analysis of ODC data provides a valuable diagnostic method for evaluating the quality of the various phases of the software life cycle (design, development, test and service) and the maturity of the product.

While techniques and tools for statistical modeling and root cause analysis have been in use for many years and in many parts of the software industry, their use in the hardware world is very limited. Experiments showed that statistical modeling, using the models developed for software reliability, can give the hardware community the same benefits they offer software reliability estimation [28]. Because the reliability requirements of a hardware system are usually higher than those of a software system, the use of quality management techniques and tools, like ODC, can provide vital assistance in the management of hardware design and verification, perhaps to a larger extent than they currently do in the software industry. We believe that the main reason for the minimal use of such tools in the hardware industry is the lack of awareness for the tools and the techniques, resulting in the use of inferior in-house techniques.

3.5 Concurrent Testing

Perhaps the most important area for emerging technologies is testing of concurrent and distributed systems. In software, the use of parallel and multi-threaded applications is becoming wide-spread, for example in server side Java programs. In hardware, multiprocessor systems are becoming common. These technologies present new testing challenges in the form of synchronization problems, races, and deadlocks. Because this technology is new, and the testing of such applications and systems especially difficult, many concurrent bugs (i.e., bugs related to races in the application) are found in the field by the end-user [29]. New technologies for testing are being developed in the two communities at a high rate. In hardware, new test generators for concurrent systems have been created [30, 31]. The challenge is to create interesting tests with multiple correct expected results and still be able to check if the tests executed correctly. In software, tools that monitor the execution and identify potential race conditions [32], as well as tools that work as irritators for multi-threaded Java applications [33] are being developed.

3.6 Automatic Test Generation

In hardware, due to the methodology used, test generation has been a core testing technology for over 20 years. The tools are mature and have very advanced capabilities. It is not trivial to adapt such tools to software testing. While most of the benefits of test generation are achieved when they are used in conjunction with behaviorals (that are discussed next), the economic benefits of such tools are undeniable.

FSM based test generators are becoming popular in software testing [34]. These test generators use advanced graph traversal algorithms to generate tests that cover an abstract FSM model of the tested system. Later, the generated tests are executed on the tested system itself, using the results from the abstract model as the expected results. FSM based test generators, originally used in hardware verification [35], made their way into testing of communication protocols that resemble state-machines, and have lately made their way to “normal” software such as file system middle-ware [14]. There are already a large number of such tools deployed, although it is not yet clear if they will become common in the industry.

3.7 Use of Behaviorals

Behaviorals, or reference models, are programs that emulate the behavior of the tested system. They are created in order to predict the expected behavior of the tested system for any given stimuli, and thus provide a means by which to check if the tested system behaved correctly. Behaviorals are essential when massive amounts of tests are generated. In hardware, behaviorals, as well as test generators are ubiquitous. In order to test hardware, many tests need to be executed and it is not feasible to create them and verify the results by hand. In software, even though automatic test generation is becoming more common, behaviorals are rare. The main reason is that to write the behavioral, one needs to write the entire application from scratch.

In many cases, this is not true. The creation of behaviorals can be much simpler than that of the real application. Behaviorals do not have to meet the non-functional requirement of the tested system, so that performance enhancing mechanisms, for example, can be ignored. The behavioral of a complicated parallel or concurrent application, can be a simple sequential program. In some cases the application itself can be used as its behavioral, for example, when it is running on a single processor instead of a parallel system. Another example is to use the application on one configuration and compare it to its behavior in a different configuration. Another method to simplify behaviorals is to abstract out parts of the requirements that are not needed for testing, like error handling.

The use of behaviorals allows increased automation of the testing. It reduces the manual checking that is needed for each test by automatically generating the expected result. We believe that development of behaviorals for software testing will increase the use of automatic test generation in testing and contribute to the testing quality.

4 Problems and Benefits of Sharing Technologies

In previous sections we discussed the similar problems faced by the software testing and hardware verification communities. We also talked about the detachments between the two communities. On the other hand, we described several technologies that crossed the boundary between the communities and became useful in both domains.

In this section, we discuss the difficulties of sharing and transferring technologies between the software testing and hardware verification communities, and the benefits that close relations between the communities can yield. As an example, we describe the Verification Technologies Department at the IBM Haifa Research Laboratory.

The Verification Technologies Department at the IBM Haifa Research Laboratory is an IBM center of competence in functional verification of hardware. More than 50 people have been working for the last twelve years on functional verification tools and methodologies [5, 6]. The work done in the department concentrates on very powerful tools that are used by a small number of users, mainly developers of processors. The tools developed in the department are based on four technologies: domain-specific test generators [5], model checking [6], functional coverage analyzers [23], and FSM based test generators [35].

During these twelve years, people from the department communicated with other IBM Haifa groups involved in projects that needed support in software testing. As hardware verification people, they claimed they had relevant technologies. However, after some considerations these technologies were never used for software testing. The main reason for this is the different environment in which tools operate. For example, a typical site that uses Genesys [5], a test generator for processors, dedicates at least one person for its support. This investment in support is possible only if Genesys is expected to be used for a long time on several projects.

While the technology of test generation used in Genesys is very relevant for testing software in specific domains such as compilers, creating a similar tool for generic software testing is quite different. The differences between projects developed by the same software development house mean that such software houses cannot invest in expensive tools that may not be applicable to the next project. Therefore, tools developed for software testing must be easy to maintain, very easy to learn, and must be affordable to many users.

Although in many cases we were able to demonstrate the benefits of using the tools and technologies developed for hardware verification, we found out that it is not possible to sell (or even give away) the tools. This is due to the different requirements for tools used in hardware verification and software testing.

As a result, we decided to create a software testing tool group under the same management as the hardware verification tool group. The mandate of the new group is to leverage our acquired knowledge of hardware verification into software testing. The new group had some initial success but still faces some difficulties.

An example of a success in transfer is a generator called Gotcha [35]. Gotcha started as processor verification tool. Later, the potential of using such technology in software testing was identified, a testing framework, named TCBeans, was developed around it [14]. Gotcha-TCBeans is now used in testing file system middleware, APIs for creating web interface for legacy applications, software for call centers, and many others.

In the same manner, we tried to adapt our functional coverage tool Comet [23] for software testing, and created Focus [24], an easy to use functional coverage tool. Comet is a heavy duty tool based on a relational database, that requires a one week course by its users before it can be used. Focus, on the other hand, is a stand alone Java tool that can be used after just one hour of training. The main design goal of Focus was to be as easy-to-use as possible. The powerful options of Comet that complicate usage were not included. Unlike Gotcha, we had very limited success in introducing Focus to the software testing community. We are not sure about the reasons for our difficulties, but we believe that the main one is that we did not work with projects that have particularly high quality requirements with regard to aspects such as robustness and reliability.

5 Conclusions

In this paper, we compared the testing and verification processes currently used in software and hardware development. We showed that despite the many similarities in the design and implementation process, the philosophies and methodologies used in the two domain are different. One of the main reasons for the different approaches taken by the hardware and software communities, is the lack of interaction and communication between management, tool developers, and users, even when both communities exist in the same organization.

We showed that cooperation between the hardware verification and software testing communities can lead to the sharing of techniques between the communities, and provided several examples of such shared techniques, including code coverage and formal methods. We pointed out several technologies that can benefit both communities in the near future or even at present, such as FSM based test generators, and techniques for testing concurrent systems.

We believe that close interaction between the hardware verification and software testing communities will greatly benefit both communities, and therefore it is important to get the two communities to exchange ideas and experience at all levels. We believe that the best way to increase the cooperation between the software testing and hardware verification communities is to create joint groups or departments that deal with both areas. Of course, this solution is possible only in organizations that are active in both areas. As an example for the potential benefits of such close cooperation, we described the Verification Technologies Department at the IBM Haifa Research Lab, that contains groups dedicated to software and hardware verification. The close relations between the groups has led to the successful transfer of several technologies and tools between the communities.

Another way to facilitate communication is through common conferences, where people from one community can learn about what is being done in the other. These conferences can be intra-organizational, industry-wide, or academic. This can be done with new conferences that combine software testing and hardware verification, and by to co-locating existing relevant conferences together.

References

[1] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison Wesley, 1995.

[2] D. A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In *Proceedings of the 5th Static Analysis Symposium,* 1998.

[3] R. Chillarege. Orthogonal defect classification. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 9. McGraw Hill, 1996.

[4] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

[5] A. Aharon, D. Goodman, M. Levinger, Y Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of the 32nd Design Automation Conference*, pages 279–285, June 1995.

[6] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: an industry-oriented formal verification tool. In *Proceedings of the 33rd Design Automation Conference*, pages 655–660, June 1996.

[7] J. Bhasker. *VHDL Primer*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1994.

[8] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.

[9] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *proceedings of STAR98: the 7th international conference on software testing analysis and review*, May 1998.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT-Press, 1999.

[11] M.R. Lyu. *The Handbook of Software Reliability Engineering*. McGraw Hill, 1996.

[12] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *International Conference on Software Engineering*, pages 257–266, 1999.

[13] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT solver, BDDs, and simulation. In *Proceedings of the 2000 International Conference on Computer Design*, pages 459–464, September 2000.

[14] I. Gronau, A. Hartman, A. Kirshin, K. Nagin, and S. Olvovsky. A methodology and architecture for automated software testing. http://www.haifa.il.ibm.com-/projects/verification/gtcb/publications.html.

[15] S. Dutta, A. Katti, M.A. Lee, and L. N. Van Wassenhove. 1997 software best practice survey: Analysis of results. 98/35/TM/RISE, INSEAD, Fontainebleau, France, 1998.

[16] D. J. Smith. *HDL Chip Design: A practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*. Doones Publications, 1996.

[17] S. Dutta, S. Kulandaiswamy, and L. N. Van Wassenhove. Benchmarking european software management best practices. *Communications of the ACM*, 41(6):77–86, 1998.

[18] Reasoning.com - InstantQA source code bug finding service, software testing tool. http://www.reasoning.com.

[19] C. D. Warner Jr. Evaluation of program testing. TR 00.1171, IBM Data Systems Devision Development Laboratories, IBM Poughkeepsie, N.Y., 1964.

[20] I. N. Hirsh. MEMMAP/360. TR-p-1168, IBM System Development Division, Product Test Laboratories, IBM Poughkeepsie, N.Y., 1967.

[21] Visual StateScore. http://www.summit-design.com-/products/vcd/index.html.

[22] R. Raghavan and J. Baumgartner. Covet: a coverage tracker for collision events in system verification. In *Proceedings of the 5th IEEE International Conference on Performance, Computing and Communications*, pages 172–177, June 1998.

[23] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[24] Focus. http://www.alphaworks.ibm.com.

[25] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 3:182–211, 1976.

[26] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[27] J.D. Musa, A. Iannino, and K Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[28] Y. Malka and A. Ziv. Design reliability - estimation through statistical analysis of bug discovery data. In *Proceedings of the 35th Design Automation Conference*, pages 644–649, June 1998.

[29] B. Lewis and D. J. Berg. *Threads Primer*. Prentice Hall, 1996.

[30] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkash, K. Holtz, A. Long, D. King, and S. Barret. A methodology for the verification of a "system on chip". In *Proceedings of the 36th Design Automation Conference*, pages 574–579, June 1999.

[31] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi. Simulation-based approach to architectural verification of multiprocessor systems. In *Proceedings of the International Phoenix Conference on Computers and Communications*, pages 34–37, June 1995.

[32] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the 2nd SIGMATRICS symposium on Parallel and Distributed Tools*, August 1998.

[33] O. Edelstein, E. Farchi, Y. Nir G Ratzaby, and S Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.

[34] E. Farchi, A. Hartman, and S. S. Pinter. Using model-based test generators to test for standard conformance. *IBM Systems Journal*, 41(3):89–110, 2002.

[35] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage driven test generation. In *Proceedings of the 36th Design Automation Conference*, pages 970–975, June 1999.