Int J Softw Tools Technol Transfer (2011) 13:201–205
DOI 10.1007/s10009-011-0194-x

INTRODUCTION

Preface

Hana Chockler · Alan J. Hu

Published online: 31 March 2011
© Springer-Verlag 2011

1 Verification: different directions and approaches

This special section consists of extended versions of some of the best papers from the fourth *Haifa Verification Conference* (*HVC*), held in 2008 October 27–30, in Haifa, Israel (see [2] for the proceedings of the conference). The HVC series had its genesis as an internal IBM formal verification workshop at the start of the millennium, but expanded rapidly into an annual, international conference, starting in 2005. As we write this preface, preparations are underway for the seventh HVC, and the conference continues to thrive.

Verification is a booming research area, and one of the areas in modern computer science where exciting research happens hand-in-hand with no less exciting application of the research results in industry. In fact, for both hardware and software systems, verification efforts are planned as an integral part of the development and deployment cycle, and the methods and tools used in industry become more and more elaborate as the systems grow in size and complexity. Testing of software and simulation-based hardware verification continuously evolve and use more and more algorithmic methods in order to improve the quality of verification. Formal verification, an area consisting of methods for mathematically proving correctness of systems with respect to formally defined properties, is now applied to all industrial chip designs. Formal verification for software, once regarded by industry with a large dose of skepticism due to the inherent complexity of modern software systems, gains acceptance as a reliable way to verify correctness of mission-critical systems and systems for which bug fixing is very expensive (such as embedded systems).

HVC is a conference specifically targeted to bringing together all different flavors of the verification domain with the goal of discovering new approaches and applications that can be extended beyond their natural domain and be applied to other verification areas. To illustrate why this is such an important goal, consider two dimensions to classify verification research: the technique used and the application domain targeted. Verification techniques have historically been classified as formal (e.g., theorem proving, model checking, formal equivalence checking, type checking) versus dynamic (e.g., hardware simulation, software test, hardware emulation), with a lot of exciting research happening as well at intermediate points of this spectrum, mixing formal and dynamic techniques. Orthogonally, the verification community has historically been split between those doing hardware verification and those doing software verification. As with the other dimension, there is also a lot of exciting research happening at intermediate points of the spectrum (e.g., verification of low-level system software, embedded software and firmware, or system-level descriptions of hardware).[1] HVC covers this entire cross-product space [formal, dynamic] × [hardware, software]. The goal of this breadth is to promote synergies, cross-fertilization, and fruitful collaboration. Real-life systems have hardware as well as software (as well as firmware, analog components, etc.), and real-life

H. Chockler (✉)
IBM Research,
Haifa, Israel
e-mail: HANAC@il.ibm.com

A. J. Hu
University of British Columbia,
Vancouver, Canada

[1] As we write this, it's interesting to note that even the terminology reflects the historical isolation of the different verification communities. For example, what the hardware community calls "simulation" corresponds to what the software community calls "test". Or, the software community often refers to formal techniques as being "static" but this term is rarely used in the hardware community.

verification needs formal as well as dynamic techniques (as well as mixtures of the two). Bringing all these communities together results in a united attack on the common challenge of verification complexity.

This is the second year that a special section dedicated to the best papers of HVC is published in STTT (see [11] for the special section dedicated to HVC 2007). As in the last year, this section shows the breadth of research in verification and fruitful cross-fertilization between different verification domains.

2 The papers

The papers selected for this special section demonstrate some of the richness and variety of verification topics at HVC. These papers were chosen among the top-rated papers by the conference program committee, and of course, given the mandate of this journal, they present significant intellectual advances along with either implementation into software tools with experimental evaluation, or at least (for the more theoretical work), specific theory in ready-to-implement form. The selected papers were all extended and improved based on feedback at the conference, and then underwent at least one additional round of full reviewing by anonymous referees. Although this process created a substantial delay between the conference and this publication of the extended journal versions, we feel that the quality is worth the wait.

Evaluating workloads using comparative functional coverage by Yoram Adler, Dale Blue, Thomas Conti, Richard Prewitt, and Shmuel Ur

This paper is about software testing, in particular, the problem of improving the coverage of testing by introducing the concept of comparative functional coverage.

Coverage is a widely used heuristic for measuring the exhaustiveness of a test suite. (For a survey of verification coverage, see e.g., [8].) The simplest form is *code coverage*, which measures the percentage of lines of codes executed by a test. For example, if we were to test the following code:

```
int abs(int i) {
 if (i<0)
  return -i;
 else
  return i;
}
```

with the testcase `abs(-1)`, for code coverage, we would report 67% statement coverage (since we exercise the if-condition and one of two return statements) and 50 branch coverage (since we exercise one of the two possible branch outcomes). Code coverage is easy to define and measure, but it is very weak: 100% (or close to it) code coverage is usually required before moving to more complex and subtle ways of measuring coverage.

Functional coverage, which is the type of coverage discussed in this paper, is one of the most important of these more complex and subtle coverage metrics. It measures the functionality exercised during a test suite, with the goal of maximizing it. The space of functionality to explore is defined by the software developers, testers, and customers. The paper addresses the fact that different customers use the same software in different ways, and hence some profiling of customers needs to be done in order to adapt the testing and coverage efforts to the most important areas for each customer.

The basic challenge is that different customers have different workloads, so a software test suite should be designed to provide the best possible coverage for each customer. Comparative functional coverage provides a means of comparing coverage under differing workloads, to identify what areas are highest priority for more testing, for a specific customer. The authors implement this concept in IBM's FoCuS tool for coverage analysis and present encouraging experimental results. The tool is available for public download.

As an aside, the target application domain for this paper is mainframe operating system software. So, although the paper is nominally about software verification, it also reflects many hardware issues relating to the underlying machine. Verification of system software is a critical issue, highlighting the importance of interaction and cross-fertilization between the hardware and the software verification communities. This paper is also perhaps the one that lends itself most naturally to wide industrial adoption.

Iterative delta debugging by Cyrille Artho

This paper is also a software verification paper, but from another angle—automated debugging of software. Automated debugging attempts to locate the reason for a software failure. *Delta debugging*, introduced by Andreas Zeller [12], is a highly regarded approach to automated debugging, which minimizes the difference between two inputs, where one input is processed correctly while the other input causes a failure, using a series of test runs to determine the outcome of applied changes. The basic intuition behind delta debugging is very simple. For example, imagine a movie file that causes a video player to crash. To isolate what part of the movie is causing the problem, one might split the movie into two halves, try each half, and continue with divide-and-conquer until isolating a short snippet of the movie that causes the crash. Delta debugging simply automates this process. Delta debugging becomes more complex when the input is structured—e.g., a bug might arise only from the interaction between something early in the input and much later in the

input—and this paper, aside from its technical contribution, also provides a literature survey of these issues and techniques for addressing them.

Delta debugging can also be applied to isolate a bug-inducing change to a program. In this paradigm, we assume a test (e.g., a regression test or test suite) on which the previous version of a program works and the current one does not. Now, we treat the difference (as produced by a utility like `diff`) between the two files as the input to delta debugging, to automatically find a small patch between the previous good version and a new version that is much closer to the previous good version, but still fails the test (or alternatively, find a small patch between the current bad version and a new "previous" version that passes the test). This is the paradigm for delta debugging used in this paper.

The preceding description assumes we have known good and bad versions of the program being debugged, but this is often not the case in practice. Worse, the testcase that fails on the current version might not run correctly on *any* older version, because of missing features in the older versions, or other bugs that masked this bug but have subsequently been fixed. This paper introduces *iterative delta debugging* to address this problem.

Iterative delta debugging starts with a test that fails on the current version of a program. It then undertakes an automated exploration of progressively older versions of the program in the version control system. If the older version also fails the test, we ignore it and check an even older version of the program. If the older version passes the test, then we have known-good and known-bad versions of the program and can proceed to normal delta debugging to isolate the cause. The interesting case is if the older version fails the test in some different manner, or crashes, as would happen if there were other bugs that had been fixed since the older version, or new features that had been added. In this case, iterative delta debugging uses delta debugging to compute a small patch to back port the features or bug fixes from the newer (known-bad) version into the older (differently-bad) version, thereby allowing the test either to pass or fail on the (patched) older version. The entire process is automatic.

The author implemented the concept in a tool, targeting several of the most popular projects on SourceForge. Although the tool is only a prototype, implemented as scripts, the paper provides many implementation details and design decisions. Experimental results show that the technique manages to localize bugs in many real-life examples. While the tool itself might not be ready for industrial use, implementing the algorithm and applying the technique on industrial-size software have the potential for making debugging much easier.

Delta debugging has had a large impact on the software testing community. It is considered so important that Andreas Zeller, one of the inventors of delta debugging, was an invited speaker at HVC in 2007 [10]. The paper on iterative delta debugging demonstrates that the subject continues to be highly relevant for testing community, and we are confident that we will see further research in this direction in the coming years.

Automatic boosting of cross-product coverage using Bayesian networks by Dorit Baras, Shai Fine, Laurent Fournier, Dan Geiger, and Avi Ziv

This paper describes a work in the domain of simulation-based hardware verification. Improving coverage of dynamic verification is as important in hardware as it is in software, but the techniques to achieve high coverage are different in each area.

Simulation-based verification of hardware creates series of stimuli for hardware designs and examines the resulting traces. Coverage in simulation-based hardware verification is measured by the percentage (of the list) of *covered events*, i.e., interesting events in the design that should be triggered by at least one of the stimuli during the verification process. The list of coverage events is created by a designer together with the verification team. Because of the concurrent nature of hardware, the list of coverage events is often generated as the cross-product (or a part of the cross-product space) of values of "interesting" signals and important events as defined by the designer. For example, when verifying a microprocessor, one might seek to test every possible combination of instruction types, combined with every possible combination of exceptions (page faults, branch mispredicts, illegal instructions, etc.). Constructing the right stimuli for a given list of coverage events is far from trivial: it requires a deep knowledge of the design in order to understand which stimuli will trigger behaviors that reach coverage events. This need for deep domain knowledge is a real challenge for simulation-based verification teams. ([9] gives an excellent, in-depth description of these issues in simulation-based hardware verification.)

This work addresses the problem of creating the right stimuli for improving coverage of verification of hardware, while eliminating the need for deep domain knowledge. The paper suggests using machine-learning techniques to automate the process of determining which input stimuli create which outputs. In particular, the paper is based on the Bayesian networks [6], a data structure for compactly representing complex conditional probabilities between variables. (The paper provides a brief introduction to Bayesian networks, for readers unfamiliar with the concept. For a more in-depth treatment, the standard text is [7].) Bayesian networks have proven capable of expressing many relationships that arise in practice, and efficient algorithms exist for learning Bayesian networks from data and then for inferring the implied probabilistic relationship between variables.

This paper introduces a Bayesian networks-based CDG (coverage directed generation) system, creating a method that enables a fully automatic construction of a CDG system without requiring any domain knowledge. The algorithms for the authors' tool are given in the paper, as well as experimental results showing the trade-offs and contributions of different heuristics and algorithms. On a real-life cross-product coverage model, the method boosts coverage in the early stages of the verification process with minimal effort.

Reducing the size of resolution proofs in linear time by Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman

The preceding papers had a dynamic verification focus, whereas this and the next paper are more oriented towards formal verification.

Formal verification is, by definition, a mathematical proof of correctness of a formal model of a system [e.g., a finite state machine (FSM)] with respect to a formally defined specification (e.g., another FSM or a temporal logic formula). There exist many algorithms and corresponding automatic tools to create such a proof—or a counter-example in case the system is incorrect with respect to the given specification. (For more information, [1] is the best-known text for these automatic techniques for formal verification.)

Currently, many of the most widely used tools rely on *SAT solvers*—programs that solve the boolean satisfiability problem—as their underlying computational engine. In the boolean satisfiability problem ("SAT" for short), we are given a propositional logic formula, traditionally in conjunctive normal form (product-of-sums), and the task is to determine a satisfying assignment, or determine that none exists. For example, the formula[2]

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c) \tag{1}$$

is satisfiable (e.g., $a = 1, b = 0, c = 1$ is one of the two satisfying assignments), whereas the formula

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg b) \wedge (\neg c) \tag{2}$$

is unsatisfiable. In general, unless $P = NP$, all algorithms for SAT require exponential time in the worst case, but modern solvers have been developed that are highly efficient on problems arising in practice, routinely handling problems with millions of variables or more.

These modern SAT solvers are based on the so-called DPLL approach (named for the authors Davis, Putnam, Logemann, and Loveland, who pioneered the general approach in the 1960s [4,3]), with recent enhancements and heuristics that make the approach efficient in practice. At their core, these solvers exhaustively search the exponential space of possible truth assignments, but gain efficiency by quickly pruning away large areas of the search space that it can deduce cannot contain solutions. If a solution is found, it is easy to check that the solution works. For example, in equation (1), it is easy to plug in the solution provided. But if the solver says the formula is unsatisfiable, as in equation (2), it is not obvious *why* it is unsatisfiable.

Fortunately, a DPLL-based SAT solver can be easily modified to produce a *resolution proof* of unsatisfiability. *Resolution* is a simple inference rule that given two clauses (whose conjunction is must be satisfiable if the formula is to be satisfiable) and a variable that appears un-negated in one clause and negated in the other, we can infer a new clause that consists of a combination of the two clauses with the variable removed. For example, if we resolve $(a \vee b)$ with $(\neg a \vee c)$, we can deduce that $(b \vee c)$, because depending on the value of a, either b or c must be true if the formula is to be satisfiable. A resolution proof of unsatisfiability is a sequence of resolution steps that show that the original clauses imply false, i.e., they are unsatisfiable. For example, returning to equation (2), we can resolve $(a \vee b)$ with $(\neg a \vee c)$ to get $(b \vee c)$. Resolving that with $(\neg b)$ yields (c). Resolving (c) with $(\neg c)$ yields false, proving unsatisfiability.

A resolution proof can be used to certify that correctness of the SAT solver's answer. More importantly in practice, it provides insight into *why* a problem is unsatisfiable, and is hence used as input to other verification algorithms (e.g., it says which variables and relationships were relevant to proving a given fact about a design). Accordingly, it is important to produce resolution proofs that are as short and simple as possible: irrelevant resolution steps should be eliminated.

This paper suggests two efficient methods for making the resolution proofs smaller. It also describes experiments with industrial hardware instances, showing that these methods significantly reduce the size of the proofs. As with the preceding paper, the paper presents all algorithms and detailed implementation notes for the authors' tool. This is the appropriate level for technology transfer, as we envision that others would re-implement the algorithms, either directly into their own SAT solvers, or as scripts tailored toward the output of those solvers.

A uniform approach to three-valued Semantics for μ-calculus on Abstractions of hybrid automata by Kerstin Bauer, Raffaella Gentilini, and Klaus schneider

HVC's scope goes beyond the dichotomy of just hardware versus software; there are many other applications of verification. This paper is in the realm of hybrid systems—systems that combine both discrete as well as continuous dynamics. Hybrid systems are extremely important in practice, with

[2] In this introduction, as in the paper we are introducing, we use mathematical logic notation, where $\vee$ denotes logical OR, $\wedge$ denotes logical AND, and $\neg$ denotes logical negation.

obvious applications in robotics and control. For example, in an aircraft control system, the discrete dynamics reflect modes of operation (e.g., take-off-go-around, autoland, etc.) or algorithmic decisions (e.g., whether to climb or descend to avoid a collision), whereas the continuous dynamics reflect the aerodynamics of an airplane in flight. Or, the verification of an industrial robot must reason about both the digital control algorithm as well as the continuous dynamics of motors, servos, sensors, and the inertia of the physical components. Less obvious—but also extremely important—applications arise because the real world is continuous, so there are always interactions between discrete and continuous at all levels of design abstraction. For example, digital electronic circuits are really analog circuits, with continuous voltages and currents obeying differential equations, but for which we have defined a digital abstraction. With modern high-performance circuits, proving that the circuit does behave as intended is a challenging hybrid system verification problem.

The standard model for hybrid systems is the *hybrid automaton*, which can be easily understood as an extension of ordinary, discrete, finite-state automata. (See e.g. [5] for a deeper presentation of the theory of hybrid automata.) Just as in normal finite-state automata, the system is assumed to contain a finite number of discrete states, with transitions among them. However, the system is also augmented with real-valued continuous variables. For example, the usual introduction of a hybrid automaton is a room with thermostat-controlled heating: the discrete states are whether the heat is on or off, and the continuous state is the room temperature. Within each state, the continuous variables obey differential equations specific to that state. In the example, when the heat is off, the room temperature will slowly decay to the outside ambient air temperature according to some differential equation; when the heat is on, the room temperature will increase, according to a different differential equation. The discrete states and transitions between them can be annotated with formulas that act as guards and invariants, allowing the continuous state to influence the discrete transitions. For example, with the thermostat, when the room temperature rises above some threshold, the system will make the discrete transition from heat-on to heat-off; when the temperature drops below some threshold, the system will make the discrete transition from heat-off to heat-on.

The hybrid automata model is very powerful and expressive, but most verification problems for them are undecidable in general. Even for highly restricted variants of hybrid automata, for which verification is decidable, the complexity is usually too high to be useful in practice. Therefore, the emphasis in hybrid systems is to develop abstractions that allow efficient verification, that do not miss bugs, but that also do not produce too many false errors (due to overly conservative abstraction). This paper combines several research directions: hybrid automata; three-valued semantics, to allow conservative approximations of both the truth and falsity of logical formulas; and abstraction, which reduces the state space of the verified system. It develops a general framework of three-valued semantics for μ-calculus (an extremely general and flexible specification language) on abstractions of hybrid automata, preserving several possible semantics of the modal operators. (The previously cited text [1] provides an introductory treatment of μ-calculus as well.) As possible special cases of the general result, the paper considers (1) modal abstractions, where the notions of may and must transitions are extended from the purely discrete to the hybrid time framework, and (2) discrete bounded bisimulation abstractions. Although this is the most theoretical of the selected papers, the ideas are readily applicable in practice, and the authors intend to implement them as a verification tool for the synchronous language Quartz.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
2. Chockler, H., Hu, A.J. (eds.): Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27–30, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5394, Springer (2009)
3. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Commun. ACM **5**(7), 394–397 (1962)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
5. Henzinger, T.: The theory of hybrid automata. In: Symposium on Logic in Computer Science (LICS), pp. 278–292 (1996)
6. Pearl, J. : Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning. UCLA Technical Report CSD-850017 (1985)
7. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, Massachusetts (1988)
8. Piziali, A.: Functional Verification Coverage Measurement and Analysis. Springer (2004)
9. Roesner, W., Wile, B., Goss, J.C.: Comprehensive Functional Verification—The Complete Industry Cycle. Morgan Kaufmann, Massachusetts (2005)
10. Yorav, K. (ed.): Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23–25, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4899, Springer (2008)
11. Yorav, K.: Haifa verification conference 2007. STTT **11**(4), 269–272 (2009)
12. Zeller, A.: Delta debugging. http://www.st.cs.uni-saarland.de/dd/