Fault Models and Test Generation for Hardware-Software Covalidation

Ian G. Harris
University of California, Irvine

Editor's note:
Mixed hardware-software systems constitute a strong paradigm shift for system validation. The main barriers to overcome are finding the right fault models and optimizing the validation flow. This article presents a research summary of these issues.

—Ahmed Jerraya, TIMA

A HARDWARE-SOFTWARE SYSTEM is one that requires the codesign and interaction of hardware and software to properly implement system functionality. Such a system can satisfy various design constraints that neither hardware nor software can meet alone. The widespread use of these systems in cost- and life-critical applications makes a systematic approach to functional verification essential. Several obstacles to verification make this a challenging problem requiring major research. Hardware verification complexity alone has increased to the point where it dominates design cost. To manage the problem's complexity, many researchers are investigating covalidation techniques, which verify functionality by simulating or emulating a system description with a given test input sequence. Other approaches verify functionality by using formal techniques, such as model checking, equivalence checking, and automatic theorem proving, to precisely evaluate a design's properties. However, covalidation's tractability makes it the only practical solution for many real designs.

Hardware-software codesign typically begins with a high-level specification and produces a partially refined design, which software compilation and behavioral hardware synthesis can complete. Covalidation occurs after each design refinement step to guarantee that synthesis has produced a correct design. If the design is correct, the synthesis process continues; otherwise, designers must repeat the previous synthesis step to correct any problems.

Covalidation has three major steps: test generation, cosimulation, and test response evaluation. The test generation process typically involves a loop that progressively evaluates and refines the test sequence until the system meets coverage goals. Cosimulation occurs next, using the resulting test sequence, followed by an evaluation of the cosimulation test responses for correctness.

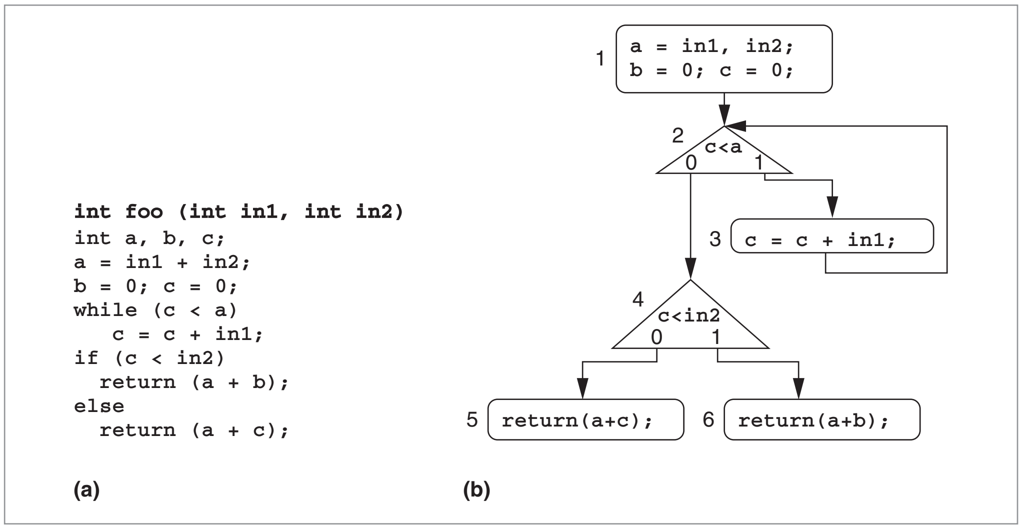
A key component of test generation is the covalidation fault model, which abstractly describes the expected faulty behaviors. The fault model provides fault detection goals for the automatic test generation (ATG) process and enables evaluation of a test sequence's fault detection qualities. This article presents a survey of test generation techniques for covalidation along with the fault models that support them.

Fault models and coverage evaluation

A design error is a difference between the designer's intent and a design's executable specification. A natural-language specification is typically used to express the designer's intent. An executable specification is a precise, simulatable description of the design. High-level hardware-software languages are often used to express executable specifications. Design errors can range from simple syntax errors confined to a single line of a design description to a fundamental misunderstanding of the design specification that can impact a large segment of the description.

A design fault describes the behavior of a set of design errors, allowing a large set of errors to be modeled by a small set of faults. A covalidation fault model defines a set of faults for an arbitrary design, enabling concise representation of a set of design errors.

Most hardware-software codesign systems use a top-down design methodology that begins with a behavioral system description. Thus, most covalidation fault models are at the behavioral level. Existing covalidation fault models vary according to the behavioral description style they use. The models originally specify system behaviors in textual languages such as VHDL and Esterel, and then convert them into an internal behavioral format for codesign and cosimulation. Various internal behavioral formats are possible.[1] The covalidation fault models currently applied to hardware-software designs originate in either the hardware or software domain.[2,3]



```
int foo (int in1, int in2)
int a, b, c;
a = in1 + in2;
b = 0; c = 0;
while (c < a)
   c = c + in1;
if (c < in2)
  return (a + b);
else
  return (a + c);
```

Figure 1. Behavioral descriptions: textual description (a), and control-dataflow graph (b).

The simple system example in Figure 1 can serve as a tool to describe covalidation fault models. Figure 1a shows a behavior, and Figure 1b shows the corresponding control-dataflow graph (CDFG). This example is limited because it comprises only a single process and contains no signals used for real-time modeling in most hardware description languages (HDLs). However, the example is sufficient to describe the relevant features of many covalidation fault models.

Textual fault models

A textual fault model is one that is directly applicable to the original textual behavioral description. The simplest textual fault model is the statement coverage metric introduced in software testing,[3] which associates a potential fault with each line of code and requires that each statement in the description execute during testing. This coverage metric has limited accuracy, partly because it ignores fault effect observation. Despite its limitations, however, designers commonly use statement coverage as a minimum test goal. Several industrial tools support the evaluation of statement coverage for hardware and hardware-software designs. Examples include Mentor Graphics' Seamless (http://www.mentor.com/seamless/), Esterel Technologies' Scade and Esterel Studio (http://www.esterel-technologies.com), Verisity's SureCov (http://www.verisity.com), and Veritable's Verity-Check (http://www.veritable.com).

Mutation analysis is a textual fault model, originally developed in the field of software test,[4] that researchers have applied to hardware validation.[5] A *mutant* is a version of a behavioral description that differs from the original by a single potential design error. A *mutation operation* is a function applied to the original program to generate a mutant. An example of a typical mutation operation is arithmetic operator replacement, which replaces each arithmetic operator with another operator. However, the mutation operations' local nature could limit their ability to describe a large set of design errors.

Control-dataflow fault models

Many covalidation fault models are based on the traversal of paths through the CDFG representing the system behavior. Applying these fault models to a hardware-software design requires converting both hardware and software components into a CDFG description. Although using these fault models with a CDFG for a single process is well understood, their use for multiple processes—such as those in a complex hardware-software system—remains poorly defined. So current codesign practice restricts the use of existing CDFG fault models to single processes. The earliest control-dataflow fault models include the branch and path coverage models used in software testing.[3]

The branch coverage metric associates potential faults with each direction of each conditional in the CDFG. Branch coverage requires that all CDFG paths covered during covalidation include both directions of all binary-valued conditionals. Several researchers have used the behavioral-validation branch coverage metric for coverage evaluation and test generation.[6,7]

The path coverage metric is more demanding than the branch coverage metric because path coverage reflects the number of control-flow paths taken. The assumption is that an error is associated with some path through the CDFG and that all control paths must execute to guarantee fault detection. The number of control paths is infinite when the CDFG contains a loop, so the path coverage metric must be used with a limit on path length.[8] Because the total number of control-flow paths grows exponentially with the number of conditional statements, several researchers have attempted to select a subset of all such paths that are sufficient for testing. Paoli, Nivet, and Santucci presented one path selection criterion,[9] based on work in software test,[10] which identifies a basic set of paths that are linearly independent and can combine to form any other path.

Laski and Korel investigated dataflow testing criteria for path selection.[11] Such testing classifies each variable occurrence as either a definition or a use. Selected paths connect a definition occurrence to a use occurrence of the same variable. For example, in Figure 1b, node 1 contains a definition of signal a, and nodes 2, 5, and 6 contain uses of signal a. Paths 1, 2, 4, 5 and 1, 2, 4, 6 must execute to cover both of these definition-use pairs. Researchers have also applied the dataflow testing criteria to behavioral hardware descriptions.[12]

The domain analysis technique in software test considers not only the control-flow path traversed but also the variable and signal values during execution.[13] A domain is a subset of a program's input space in which every element causes the program to follow a common control path. A domain fault causes program execution to switch to an incorrect domain. Researchers have applied this idea to develop a domain coverage fault model applicable to hardware and software descriptions.[14]

Many CDFG fault models consider the requirements for fault activation without explicitly considering fault effect observability. Observability-based behavioral fault models can alleviate this weakness.[15,16] The approach presented by Fallah, Devadas, and Keutzer inserts faults called tags at each variable assignment;[15] these tags represent a positive or negative offset from the correct signal value. Observability analysis along a control-flow path occurs probabilistically using the operations' algebraic properties along the path and simulation data.

State-machine fault models

Finite-state machines (FSMs) constitute the classic method of describing a sequential system's behavior, and researchers have applied fault models to state machines. The commonly used fault models are state coverage (which requires that all states be reached) and transition coverage (which requires that all transitions be traversed). Some researchers have applied state-machine transition tours—paths covering all the machine's transitions—to microprocessor validation.[17] Geist et al. proposed a user-refined transition coverage model. It selects only transitions affecting state variables identified by the user as important for test.[18] In a thorough survey article, Lee and Yannakakis use classical switching theory to explain many problems associated with state machine testing.[19]

The most significant problem with the use of state-machine fault models is the complexity resulting from the state space size of typical systems. To alleviate this problem, researchers have identified a subset of the state machines critical for validation. The Extended Finite State Machine (EFSM) and the Extracted Control Flow Machine (ECFM) create a reduced state machine by partitioning the state bits between control and data.[20,21] Bergmann and Horowitz generated a reduced state machine by projecting the original state machine onto a set of states identified as interesting for validation purposes.[22]

Gate-level fault models

A gate-level fault model is one that was originally developed for, and applied to, gate-level circuits. Manufacturing test research defines several gate-level fault models that now apply to the behavioral level.[23] For example, the stuck-at fault model assumes that an error holds each signal to a constant value of 0 or 1. Researchers have also applied the stuck-at fault model at the behavioral level for manufacturing test and for hardware-software covalidation.[24,25] Behavioral designs often represent variables with many bits and typically apply gate-level fault models to each bit individually.

Application-specific fault models

To justify the cost of developing and evaluating an application-specific fault model, the market must be very large, and the application's fault modes, well understood. For this reason, application-specific fault models appear in microprocessor test and validation.[26-31] Early microprocessor fault models target relatively generic microprocessor features. For example, researchers define a fault model for instruction-sequencing functions by describing the fault effects (activation of erroneous micro-orders)[28] and describing the fault detection requirements. More recent fault models target modern processor features such as pipelining.[30,31]

Another alternative is to let the designer define the fault model. This option relies on the designer's expertise at expressing the fault model's characteristics. Several tools automatically evaluate user-specified properties during simulation to identify faults. The simplest techniques in common hardware-software debuggers let the user specify breakpoints based on a subset of state variables. More sophisticated tools let the designer use temporal logic primitives to express faulty conditions.[32]

Interface faults

To manage the high complexity of hardware-software design and covalidation, researchers have tried to separate each component's behavior from that of the communication architecture.[33] Interface covalidation becomes more significant with the onset of core-based design methodologies that use predesigned, preverified cores. Because each core component is preverified, the system covalidation problem focuses on the interface between the components. Panigrahi, Taylor, and Dey presented a case study on interface-based covalidation of an image compression system.[34]

Additional interface complexity arises from the use of multiple clock domains in large systems. The interfaces between these different clock domains must be essentially asynchronous. Without a high-overhead, timing-independent circuit implementation (such as differential cascode voltage switch logic), asynchronous interfaces are particularly vulnerable to timing-induced faults, which cause definition of a signal value to occur earlier or later than expected.[35] The Verity-Check tool from Veritable also targets synchronization problems between multiple clock domains.

Automatic test generation techniques

There are several ATG approaches, varying according to the class of search algorithm, the fault model, the search space technique, and the design abstraction level. Performing test generation for the entire system requires uniformly describing both hardware and software component behaviors. Although many behavioral formats are possible,[1] previous ATG approaches focused on CDFG and FSM behavioral models.

Researchers have explored two classes of search algorithms: fault directed and coverage directed. Fault-directed techniques successively target a specific fault and construct a test sequence to detect it. The algorithm merges each new test sequence with the current test sequence, typically by using concatenation, and evaluates the resulting fault coverage to determine if test generation is complete. Fault-directed algorithms are complete in that they will find a test sequence for a fault if such a sequence exists, assuming they have enough CPU time.

Coverage-directed algorithms improve coverage without targeting any specific fault. The algorithm heuristically modifies an existing test set to improve total coverage and then evaluates the fault coverage produced by the modified test set. If the modified test set corresponds to an improvement in fault coverage, the algorithm accepts it. Otherwise, the algorithm either rejects the modification or uses another heuristic to determine its acceptability.

Fault-directed techniques

Several researchers have addressed the test generation problem directly at the CDFG level by identifying a set of mathematical constraints on the system inputs that cause traversal of a chosen CDFG path. With these constraints identified, the test generation problem is equivalent to the problem of solving the constraints simultaneously to produce a test sequence at the system inputs. The test algorithm can associate each CDFG path with a set of constraints that must be satisfied to cause path traversal. For example, in Figure 1b, the path containing nodes 1, 2, 4, and 6 pertains to the requirement that $c \geq a$ and $c < in_2$. Because the operations in a hardware-software description can be either Boolean or arithmetic, the solution method chosen must handle both types of operations. The Boolean version of the problem, traditionally called the satisfiability (SAT) problem, is a well-studied, fundamental, nondeterministic-polynomial-time complete (NP-complete) problem. Handling both Boolean and arithmetic operations is challenging because researchers have presented separate solutions to the two problems. For example, techniques based on binary decision diagrams (BDDs) perform well for Boolean operations, but the complexity of modeling word-level operations with BDDs is high.

Researchers have defined the HSAT problem as a hybrid version of the SAT problem; HSAT considers linear arithmetic constraints along with Boolean SAT constraints.[36,37] Fallah, Devadas, and Keutzer presented an algorithm[36] to solve the HSAT problem that combines a SAT-solving technique with a traditional linear program solver.[38] The algorithm progressively selects variables and explores value assignments while maintaining consistency between Boolean and arithmetic domains. Other researchers have solved the problem by expressing all constraints in a single domain and using a solver for that domain. Zeng, Kalla, and Ciesielski formulated Boolean SAT constraints as integer linear arithmetic constraints.[39]

Constraint logic programming (CLP) techniques can handle a broad range of constraints, including nonlinear constraints on both Boolean and arithmetic variables.[40] CLP techniques are novel in their use of rapid, incremental consistency checking to avoid exploring invalid parts of the solution space. Different CLP solvers use various constraint description formats to capture complex constraints. Researchers have used the GNU Prolog engine to generate tests by converting Boolean and arithmetic constraints into Prolog predicates.[41,42] Others have used CLP to generate tests for path coverage in a CDFG,[8] where arithmetic constraints expressed at each branch point of a path are solved simultaneously to generate a test that traverses the path. Similar approaches explore a subset of linearly independent paths.[9] Xin and I used the CLP approach to generate tests related to the synchronization between concurrent hardware-software processes.[43]

Although, by their nature, BDDs represent Boolean functions, they can describe the CDFG of a behavioral-level VHDL description.[44] Such approaches describe arithmetic functions in the Boolean domain by describing each output bit function as a BDD. These approaches identify test patterns by solving the SAT problem for the machine that is the exclusive-OR of the good and faulty machines.

Ho et al. accomplished state machine testing by defining a *transition tour*, a path that traverses each state machine transition at least once.[17] They generated transition tours by iteratively improving an existing partial tour, concatenating onto it the shortest path to an uncovered transition. Geist et al. generated a test sequence for each transition by asserting that a given transition does not exist in a state machine model,[18] and then using Carnegie Mellon University's Symbolic Model Verifier (SMV) tool to disprove the assertion.[45] A by-product of disproving the assertion is a test sequence that includes the transition.

If a fault effect is directly observable at the machine outputs, then covering each state and transition during test is sufficient for observing the fault. However, a fault effect could cause the machine to be in an incorrect state that is not immediately observable at the outputs. In this case, it would be necessary to apply a distinguishing sequence to differentiate each state from all other states, based on the output values. The testing problems associated with state machines, including the identification of distinguishing, synchronizing, and homing sequences, are well defined.[19]

A significant limitation to test generation techniques for state machines is the complexity of the state enumeration process performed during test generation. The abstraction method for representing the state machine greatly impacts this complexity. BDDs can represent the state transition relation and efficiently perform implicit state enumeration by defining an image computation, which computes the states that are reachable from a given set of states.[46] The efficiency of this state enumeration method has led to its use during the state-machine test generation process.[21,22,47]

Coverage-directed techniques

Several techniques generate test sequences without targeting any specific fault. Such techniques improve coverage by modifying an existing test sequence and then evaluating the coverage of the new sequence. These techniques differ with regard to the methods they use to modify the test sequence, the cost function for evaluating the new sequence, and the criteria for accepting it. The modification method is typically either random or directed random.

Corno et al. presented such a technique, using a genetic algorithm to successively improve the population of test sequences.[7] In the terminology of genetic algorithms, a *chromosome* describes a test sequence. Many test sequences are initially generated randomly. Random matings can occur between the chromosomes that describe the test sequences, but the mating process defines and restricts the way the two test sequences merge. The cost (or fitness) function to evaluate a test sequence is the total number of elementary operations (variable read/write) executed.

Lajolo et al. used a random-mutation hill-climbing (RMHC) algorithm to randomly modify a test sequence and improve a testability cost function.[48] The criteria for accepting a new sequence is whether it improves the cost function. This approach targets the single stuck-at fault model applied to the individual bits of each variable in the behavioral description. The cost function contains two parts: the number of statements that the sequence executes, and the number of outputs containing a fault effect.

Yuan et al. generated directed-random test pattern sequences.[49] This approach assumes no particular fault model, so the user must provide the directives for pattern generation. The two types of directives are constraints (which define boundaries for the space of feasible test patterns) and biases (which nonrandomly direct value assignments to signals). Test engineers must develop a set of constraints and biases that will reveal a particular class of faults.

THE FIELD OF hardware-software covalidation is maturing, and researchers are beginning to agree on the essential problems they need to solve. Industrial tools provide practical solutions to test generation, particularly at the state machine level. Although automation tools are available, designers do not fully trust them; hence, considerable manual test generation is still prevalent in the vast majority of design projects.

A significant obstacle to widespread industrial acceptance of available covalidation techniques is mistrust in the correlation between covalidation fault models and real design errors. Some ATG techniques are applicable to large-scale designs, but designers will not use them without having confidence in the underlying fault models. Evaluating fault models requires identifying a correlation between fault coverage and the detection of real design errors. The compilation of design errors by designers is essential to this evaluation. Available research in this direction should be used to evaluate existing covalidation fault models.[50] With the empirical evaluation of covalidation fault models, automation of test generation will lead to large increases in covalidation productivity.

Much of the research in hardware-software covalidation builds on previous research in the hardware and software domains, but communication between hardware and software components is a problem unique to hardware-software covalidation. The interfaces between hardware and software introduce many new design issues that can cause errors. For example, complex systems often use an asynchronous communication protocol implemented in both hardware and software. Such asynchronous communication is difficult for both hardware and software designers, and so might lead to many design errors.

Hardware-software communication complexity also increases because hardware and software handle interprocessor communication in different ways. HDLs typically provide only the most basic synchronization mechanisms, such as VHDL's wait expression. More-complicated protocols require manual implementation and are therefore vulnerable to design errors. Interprocess communication in software tends to use high-level communication primitives such as monitors (Java's synchronized statement, for example). Although each primitive's implementation is correct, the designer could use the primitive incorrectly, resulting in design errors. More research investigating the testing of interfaces between hardware and software components is essential. ■

Acknowledgments

This work was supported in part by the National Science Foundation under grant number 0204134.

■ References

1. D.D. Gajski and F. Vahid, "Specification and Design of Embedded Hardware-Software Systems," *IEEE Design & Test of Computers*, vol. 12, no. 1, Spring 1995, pp. 53-67.
2. S. Tasiran and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," *IEEE Design & Test of Computers*, vol. 18, no. 4, July-Aug. 2001, pp. 36-45.
3. B. Beizer, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, 1990.
4. K.N. King and A.J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software Practice and Eng.*, vol. 21, no. 7, July 1991, pp. 685-718.
5. G. Al Hayek and C. Robach, "From Specification Validation to Hardware Testing: A Unified Method," *Proc. Int'l Test Conf.* (ITC 96), IEEE Press, 1996, pp. 885-893.
6. A. von Mayrhauser et al., "On Choosing Test Criteria for Behavioral Level Hardware Design Verification," *Proc. IEEE Int'l High-Level Design Validation and Test Workshop* (HLDVT 00), IEEE CS Press, 2000, pp. 124-130.
7. F. Corno et al., "Automatic Test Bench Generation for Validation of RT-Level Descriptions: An Industrial Experience," *Proc. Design, Automation and Test in Europe* (DATE 00), IEEE CS Press, 2000, pp. 385-389.
8. R. Vemuri and R. Kalyanaraman, "Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint Programming," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 2, June 1995, pp. 201-214.
9. C. Paoli, M.-L. Nivet, and J.-F. Santucci, "Use of Constraint Solving in Order to Generate Test Vectors for Behavioral Validation," *Proc. IEEE Int'l High-Level Design Validation and Test Workshop* (HLDVT 00), IEEE CS Press, 2000, pp. 15-20.
10. T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 12, Dec. 1976, pp. 308-320.

11. J.W. Laski and B. Korel, “A Data Flow Oriented Program Testing Strategy,” *IEEE Trans. Software Eng.*, vol. 9, no. 3, May 1983, pp. 347-354.
12. Q. Zhang and I.G. Harris, “A Data Flow Fault Coverage Metric for Validation of Behavioral HDL Descriptions,” *Proc. Int’l Conf. Computer-Aided Design* (ICCAD 00), ACM Press, 2000, pp. 369-372.
13. L. White and E. Cohen, “A Domain Strategy for Computer Program Testing,” *IEEE Trans. Software Eng.*, vol. 6, no. 3, May 1980, pp. 247-257.
14. Q. Zhang and I.G. Harris, “A Domain Coverage Metric for the Validation of Behavioral VHDL Descriptions,” *Proc. Int’l Test Conf.* (ITC 00), IEEE Press, 2000, pp. 302-308.
15. F. Fallah, S. Devadas, and K. Keutzer, “OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification,” *Proc. Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 152-157.
16. P.A. Thaker, V.D. Agrawal, and M.E. Zaghloul, “Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test,” *Proc. VLSI Test Symp.* (VTS 99), IEEE CS Press, 1999, pp. 182-188.
17. R.C. Ho et al., “Architecture Validation for Processors,” *Proc. 22nd Ann. Int’l Symp. Computer Architecture* (ISCA 95), ACM Press, 1995, pp. 404-413.
18. D. Geist et al., “Coverage-Directed Test Generation Using Symbolic Techniques,” *Proc. 1st Int’l Conf. Formal Methods in Computer-Aided Design* (FMCAD 96), *Lecture Notes in Computer Science*, vol. 1166, Springer, 1996, pp. 143-158.
19. D. Lee and M. Yannakakis, “Principles and Methods of Testing Finite State Machines—A Survey,” *Proc. IEEE Trans. Computers*, vol. 84, no. 8, Aug. 1996, pp. 1090-1123.
20. K.-T. Cheng and A.S. Krishnakumar, “Automatic Functional Test Bench Generation Using the Extended Finite State Machine Model,” *Proc. Design Automation Conf.* (DAC 93), ACM Press, 1993, pp. 1-6.
21. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, “Abstraction Techniques for Validation Coverage Analysis and Test Generation,” *IEEE Trans. Computers*, vol. 47, no. 1, Jan. 1998, pp. 2-14.
22. J.P. Bergmann and M.A. Horowitz, “Improving Coverage Analysis and Test Generation for Large Designs,” *Proc. Int’l Conf. Computer-Aided Design* (ICCAD 99), ACM Press, 1999, pp. 580-583.
23. M.L. Bushnell and V.D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Kluwer Academic, 2000.
24. P.A. Thaker, V.D. Agrawal, and M.E. Zaghloul, “Register-Transfer Level Fault Modeling and Test Evaluation Techniques for VLSI Circuits,” *Proc. Int’l Test Conf.* (ITC 00), IEEE Press, 2000, pp. 940-949.
25. A. Fin et al., “SystemC: A Homogenous Environment to Test Embedded Systems,” *Proc. 9th Int’l Symp. Hardware/Software Codesign* (CODES 01), ACM Press, 2001, pp. 17-22.
26. M. Puig-Medina, G. Ezer, and P. Konas, “Verification of Configurable Processor Cores,” *Proc. Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 426-431.
27. J. Shen and J.A. Abraham, “Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation,” *Proc. Int’l Test Conf.* (ITC 98), IEEE Press, pp. 990-999.
28. D. Brahme and J.A. Abraham, “Functional Testing of Microprocessors,” *IEEE Trans. Computers*, vol. 33, no. 6, June 1984, pp. 475-485.
29. A.J. van de Goor and Th. J.W. Verhallen, “Functional Testing of Current Microprocessors,” *Proc. Int’l Test Conf.* (ITC 92), IEEE Press, 1992, pp. 684-695.
30. P. Mishra, N. Dutt, and A. Nicolau, “Automatic Verification of Pipeline Specifications,” *Proc. 6th IEEE Int’l High-Level Design Validation and Test Workshop* (HLDVT 01), IEEE CS Press, 2001, pp. 9-13.
31. N. Utamaphetai, R.D. Blanton, and J.P. Shen, “Relating Buffer-Oriented Microarchitecture Validation to High-Level Pipeline Functionality,” *Proc. 6th IEEE Int’l High-Level Design Validation and Test Workshop* (HLDVT 01), IEEE CS Press, 2001, pp. 3-8.
32. R. Grinwald et al., “User Defined Coverage—A Tool Supported Methodology for Design Verification,” *Proc. Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 158-163.
33. J.A. Rowson and A. Sangiovanni-Vincentelli, “Interface-Based Design,” *Proc. Design Automation Conf.* (DAC 97), ACM Press, 1997, pp. 178-183.
34. D. Panigrahi, C.N. Taylor, and S. Dey, “Interface Based Hardware/Software Validation of a System-on-Chip,” *Proc. IEEE Int’l High-Level Design Validation and Test Workshop* (HLDVT 00), IEEE CS Press, 2000, pp. 53-58.
35. Q. Zhang and I.G. Harris, “A Validation Fault Model for Timing-Induced Functional Errors,” *Proc. Int’l Test Conf.* (ITC 01), IEEE Press, 2001, pp. 813-820.
36. F. Fallah, S. Devadas, and K. Keutzer, “Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability,” *Proc. Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 528-533.
37. F. Fallah, A. Pranav, and S. Devadas, “Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage,” *Proc. Design Automation Conf.* (DAC 99), 1999, pp. 666-671.

38. T. Larrabee, “Test Pattern Generation Using Boolean Satisfiability,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, Jan. 1992, pp. 4-15.
39. Z. Zeng, P. Kalla, and M. Ciesielski, “LPSAT: A Unified Approach to RTL Satisfiability,” *Proc. Design, Automation and Test in Europe* (DATE 01), IEEE CS Press, 2001, pp. 398-402.
40. P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.
41. Z. Zeng, M. Ciesielski, and B. Rouzeyre, “Functional Test Generation Using Constraint Logic Programming,” *Proc. 11th IFIP VLSI-SOC Conf.*, Int’l Federation for Information Processing, 2001, pp. 375-387.
42. D. Diaz, *GNU Prolog: A Native Prolog Compiler with Constraint Solving over Finite Domains*, Ed. 1.7 for GNU Prolog version 1.2.16, 2002; http://pauillac.inria.fr/~diaz/gnu-prolog/manual/.
43. F. Xin and I.G. Harris, “Test Generation for Hardware-Software Covalidation Using Non-Linear Programming,” *Proc. 7th Ann. IEEE Int’l High-Level Design Validation and Test Workshop* (HLDVT 02), IEEE CS Press, 2002.
44. F. Ferrandi et al., “Functional Test Generation for Behaviorally Sequential Models,” *Proc. Design, Automation and Test in Europe* (DATE 01), IEEE CS Press, 2001, pp. 403-410.
45. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic, 1993.
46. H.J. Touati et al., “Implicit State Enumeration of Finite State Machines Using BDDs,” *Proc. Int’l Conf. Computer-Aided Design* (ICCAD 90), IEEE CS Press, 1990, pp. 130-133.
47. S. Sheng and M.S. Hsiao, “Efficient Preimage Computation Using a Novel Success-Driven ATPG,” *Proc. Design, Automation and Test in Europe* (DATE 03), IEEE CS Press, 2003, pp. 822-827.
48. M. Lajolo et al., “Behavioral-Level Test Vector Generation for System-on-Chip Designs,” *Proc. IEEE Int’l High-Level Design Validation and Test Workshop* (HLDVT 00), IEEE CS Press, 2000, pp. 21-26.
49. J. Yuan et al., “Modeling Design Constraints and Biasing in Simulation Using BDDs,” *Proc. Int’l Conf. Computer-Aided Design* (ICCAD 99), ACM Press, 1999, pp. 584-589.
50. S. Tasiran and K. Keutzer, “Coverage Metrics for Functional Validation of Hardware Designs,” *IEEE Design & Test of Computers*, vol. 18, no. 4, July-Aug. 2001, pp. 36-45.

Ian G. Harris is an assistant professor at the School of Information and Computer Science at the University of California, Irvine. His research interests include hardware-software covalidation; FPGA test; and the test of globally asynchronous, locally synchronous systems. He has a BS in computer science from the Massachusetts Institute of Technology; and an MS and PhD in computer science, both from the University of California, San Diego. After receiving his PhD, he worked as an assistant professor at the University of Massachusetts, Amherst. He is a member of the IEEE Computer Society.

■ Direct questions and comments about this article to Ian G. Harris, School of Information and Computer Science, University of California, Irvine, CA 92697; harris@ics.uci.edu.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.

Moving?

Please notify us four weeks in advance

Name (Please print)

New Address

City

State/Country Zip

Mail to:
IEEE Computer Society
Circulation Department
PO Box 3014
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314

- List new address above.
- This notice of address change will apply to all IEEE publications to which you subscribe.
- If you have a question about your subscription, place label here and clip this form to your letter.

ATTACH LABEL HERE