Fault Models and Test Generation for Hardware-Software Covalidation *

Ian G. Harris
Department of Information and Computer Science
University of California
Irvine, CA 92697
harris@ics.uci.edu

Keywords: Hardware-Software Covalidation, Design Validation, Test Generation, Fault Models

Abstract

The increasing use of hardware-software systems in cost-critical and life-critical applications has led to heightened significance of design correctness of these systems. This article presents a summary of research in test generation and fault models to support *hardware-software covalidation*. The covalidation problem involves the verification of design correctness using simulation-based techniques. The article focuses on the test generation process for hardware-software systems and the fault models which support test generation.

1 Introduction

A hardware-software system can be defined as one in which hardware and software must be designed together, and must interact to properly implement system functionality. By using hardware and software together, it is possible to satisfy varied design constraints which could not be met using either technology separately. The widespread use of these systems in cost-critical and life-critical applications motivates the need for a systematic approach to verify functionality. Several obstacles to the verification of hardware-software systems make this a challenging problem, necessitating a major research effort. Hardware verification complexity alone has increased to the point that it dominates the cost of design. In order to manage the complexity of the problem, many researchers are investigating *covalidation* techniques, in which functionality is verified by simulating (or emulating) a system description with a given test input sequence. In contrast, formal verification techniques have been explored which verify functionality by using formal techniques (i.e. model checking, equivalence checking, automatic theorem proving) to precisely evaluate properties of the design. The tractability of covalidation makes it the only practical solution for many real designs.

Hardware-software codesign typically starts with a high-level specification and produces a partially refined design which can be completed by software compilation and behavioral hardware synthesis. Covalidation is performed after each design refinement step to guarantee that synthesis has produced a correct design. If the design is correct then the synthesis process continues, otherwise the previous synthesis step must be iterated to correct any problems. Covalidation involves three major steps, **test generation**, **cosimulation**, and **test response evaluation**. The test generation process typically involves a loop in which the test sequence is progressively evaluated and refined until coverage goals are met. Cosimulation is then performed using the resulting test sequence, and the cosimulation test responses are evaluated for correctness. A key component of test

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

generation is the **covalidation fault model** which abstractly describes the expected faulty behaviors. The fault model is needed provide fault detection goals for the automatic test generation process, and to enable the fault detection qualities of a test sequence to be evaluated.

This article is a survey of test generation techniques for covalidation and the fault models which support them. We describe covalidation fault models in Section 2. Section 3 summarizes automatic test generation techniques which have been used for covalidation. Conclusions and future directions of the field are discussed in Section 4.

2 Fault Models and Coverage Evaluation

A *design error* is a difference between the designer's intent and an executable specification of the design. The designer's intent is most commonly expressed as a natural language specification. An executable specification is a precise description of the design which can be simulated. Executable specifications are often expressed using high-level hardware-software languages. Design errors may range from simple syntax errors confined to a single line of a design description, to a fundamental misunderstanding of the design specification which may impact a large segment of the description. The number of potential design errors is too large to be managed either automatically or manually, so a method is needed to reduce complexity without sacrificing accuracy. A *design fault* describes the behavior of a set of design errors, allowing a large set of design errors to be modeled by a small set of design faults. A *covalidation fault model* describes the definition of a set of faults for an arbitrary design. A covalidation fault model enables the concise representation of a set of design errors for an arbitrary design.

The majority of hardware-software codesign systems are based on a top-down design methodology which begins with a behavioral system description. As a result, the majority of covalidation fault models are behavioral-level fault models. Existing covalidation fault models can be classified by the style of behavioral description upon which the models are based. System behaviors are originally specified in textual languages, such as VHDL and ESTEREL, and are converted into an internal behavioral format for use in codesign and cosimulation. Many different internal behavioral formats are possible [1]. The covalidation fault models currently applied to hardware-software designs have their origins in either the hardware [2] or the software [3] domains. As a tool to describe covalidation fault models we will use the simple system example shown in Figure 1. Figure 1a shows a behavior, and Figure 1b shows the corresponding control-dataflow graph (CDFG). The example in Figure 1 is limited because it is composed of only a single process and it contains no signals which are used to model real time in most hardware description languages. In spite of these limitations, the example is sufficient to describe the relevant features of many covalidation fault models.

2.1 Textual Fault Models

A textual fault model is one which is applied directly 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 be executed during testing. This coverage metric is accepted as having limited accuracy in part because fault effect observation is ignored. In spite of its limitations, statement coverage is well used in practice as a minimal testing goal. A number of industrial tools support the evaluation of statement coverage for hardware and hardware-software designs including Mentor's Seamless tool (http://www.mentor.com), Esterel Technologies' SCADE and

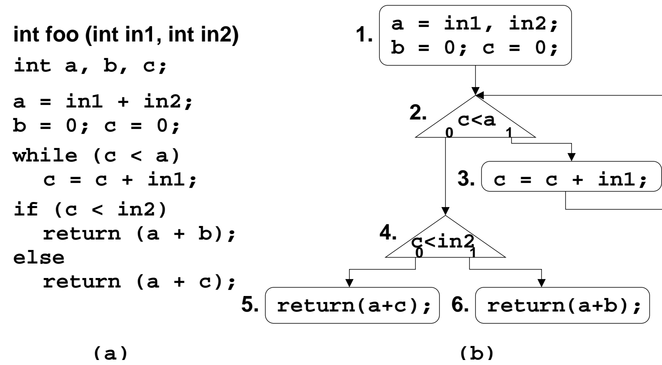

Figure 1: Behavioral Descriptions, (a) Textual Description, (b) Control-Dataflow Graph (CDFG)

Esterel Studio tools (http://www.esterel-technologies.com), Verisity's SureCov tool (http://www.verisity.com), and Veritable's Verity-Check tool (http://www.veritable.com).

Mutation analysis is a textual fault model which was originally developed in the field of software test [4], and has also been applied to hardware validation [5]. A *mutant* is a version of a behavioral description which differs from the original by a single potential design error. A *mutation operator* is a function which is applied to the original program to generate a mutant. A typical mutation operation is *Arithmetic Operator Replacement (AOR)*, which replaces each arithmetic operator with another operator. The local nature of the mutation operations may limit their ability to describe a large set of design errors.

2.2 Control-Dataflow Fault Models

A number of covalidation fault models are based on the traversal of paths through the CDFG representing the system behavior. In order to apply these fault models to a hardware-software design, both hardware and software components must be converted into a CDFG description. Applying these fault models to the CDFG representing a single process is a well understood task. Existing CDFG fault models are restricted to the testing of single processes. The earliest control-dataflow fault models include the branch coverage and path coverage [3] models used in software testing.

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

The path coverage metric is a more demanding metric 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 control flow graph and all control paths must be executed to guarantee fault detection. The number of control paths can be infinite when the CDFG contains a loop as in Figure 1b, so the path coverage metric may be used with a limit on path length [8]. Since 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 control-flow paths which are sufficient for testing. One path selection criterion is presented in [9] (based on work in software test [10]) identifies a *basis set* of paths, a subset of paths which are linearly independent and can be composed to form any other path. Previous work in software test [11] has investigated *dataflow testing* criteria for path

selection. In dataflow testing, each variable occurrence is classified as either a definition occurrence or a use occurrence. Paths are selected which 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. In this example, paths 1, 2, 4, 5 and 1, 2, 4, 6 must be executed in order to cover both of these definition-use pairs. The dataflow testing criteria have also been applied to behavioral hardware descriptions [12].

The domain analysis technique in software test [13, 3] considers not only the control-flow path traversed, but also the variable and signal values during execution. A **domain** is a subset of the input space of a program 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 which can be applied to hardware and software descriptions [14].

Many CDFG fault models consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral fault models [15, 16] to alleviate this weakness. The approach presented in [15] inserts faults called *tags* at each variable assignment which represent a positive or negative offset from the correct signal value. Observability analysis along a control-flow path is done probabilistically by using the algebraic properties of the operations along the path and simulation data.

2.3 State Machine Fault Models

Finite state machines (FSMs) are the classic method of describing the behavior of a sequential system and fault models have been defined to be applied 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. State machine *transition tours*, paths covering each transition of the machine, are applied to microprocessor validation [17]. A user-refined transition coverage model has been proposed [18] which selects only transitions which affect state variables which are identified by the user as being important for test. Many of the problems associated with state machine testing are understood from classical switching theory and are summarized in a thorough survey of 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. Several efforts have been made to alleviate this problem by identifying a subset of the state machine which is critical for validation. The Extended Finite State Machine (EFSM) [20] and the Extracted Control Flow Machine (ECFM) [21] models create a reduced state machine by partitioning the state bits between control and data bits. In [22] a reduced state machine is generated by projecting the original state machine onto a set of states which are identified as being interesting for validation purposes.

2.4 Gate-Level Fault Models

A gate-level fault model is one which was originally developed for and applied to gate-level circuits. Manufacturing testing research has defined several gate-level fault models which are now applied at the behavioral level [23]. For example, the stuck-at fault model assumes that each signal may be held to a constant value of 0 or 1 due to an error. The stuck-at fault model has also been applied at the behavioral level for manufacturing test [24] and for hardware-software covalidation [25]. Behavioral

designs often use variables which are represented with many bits, and gate-level fault models are typically applied to each bit, individually.

2.5 Application-Specific Fault Models

A fault model which is designed to be generally applicable to arbitrary design types may not be as effective as a fault model which targets the behavioral features of a specific application. To justify the cost of developing and evaluating an application-specific fault model, the market for the application must be very large and the fault modes of the application must be well understood. For this reason, application-specific fault models are seen in microprocessor test and validation [26, 27, 28, 29, 30, 31]. Early microprocessor fault models target relatively generic microprocessor features. For example, researchers define a fault model for instruction-sequencing functions [28] by describing the fault effects (i.e. activation of erroneous microorders), and describing the fault detection requirements. More recent fault models target the modern processor features such as pipelining [30, 31].

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

2.6 Interface Faults

To manage the high complexity of hardware-software design and covalidation, efforts have been made to separate the behavior of each component from the communication architecture [33]. Interface covalidation becomes more significant with the onset of core-based design methodologies which utilize pre-designed, pre-verified cores. Since each core component is pre-verified, the system covalidation problem focuses on the interface between the components.

A case study of the interface-based covalidation of an image compression system has been presented [34]. Researchers classify the interface faults which occur during the design process into three groups. Additional interface complexity is introduced by the use of multiple clock domains in large systems. The interfaces between different clock domains must be essentially asynchronous. Unless a high-overhead timing-independent circuit implementation is used (such as differential cascode voltage switch logic), asynchronous interfaces are particularly vulnerable to timing-induced faults. Timing-induced faults are described in [35] as faults which cause the definition of a signal value to occur earlier or later than expected. The Verity-Check tool from Veritable (http://www.veritable.com) also targets synchroniztion problems between multiple clock domains.

3 Automatic Test Generation Techniques

Several automatic test generation (ATG) approaches have been developed which vary in the class of search algorithm used, the fault model assumed, the search space technique used, and the design abstraction level used. In order to perform test generation

for the entire system, both hardware and software component behaviors must be described in a uniform manner. Although many behavioral formats are possible [1], previous ATG approaches have focused on CDFG and FSM behavioral models.

Two classes of search algorithms have been explored, *fault directed* and *coverage directed*. Fault directed techniques successively target a specific fault and construct a test sequence to detect that fault. Each new test sequence is merged with the current test sequence (typically through concatenation) and the resulting fault coverage is evaluated to determine if test generation is complete. Fault directed algorithms have the advantage that they are complete in the sense that a test sequence will be found for a fault if a test sequence exists, assuming that sufficient CPU time is allowed. Coverage directed algorithms seek to improve coverage without targeting any specific fault. These algorithms heuristically modify an existing test set to improve total coverage, and then evaluate the fault coverage produced by the modified test set. If the modified test set corresponds to an improvement in fault coverage then the modification is accepted. Otherwise the modification is either rejected or another heuristic is used to determine the acceptability of the modification.

3.1 Fault Directed Techniques

Several researchers have chosen to address the test generation problem directly at the CDFG level by identifying a set of mathematical constraints on the system inputs which cause a chosen CDFG path to be traversed. Once the constraints have been identified, the test generation problem is equivalent to the problem of solving the constraints simultaneously to produce a test sequence at the system inputs. Each CDFG path can be associated with a set of constraints which must be satisfied to traverse the path. For example, in Figure 1b the path containing nodes 1, 2, 4, and 6 is associated with the requirement that $c \geq a$ and $c < in2$. Because the operations found in a hardware-software description can be either boolean or arithmetic, the solution method chosen must be able to handle both types of operations. The boolean version of the problem is traditionally referred to as the SATISFIABILITY (SAT) problem and has been well studied as the fundamental NP-complete problem. Handling both boolean and arithmetic operations poses an efficiency problem because classical solutions to the two problems have been presented separately. For instance, BDD-based techniques perform well for boolean operations but the complexity of modeling word-level operations with BDDs is high.

In [36, 37] researchers define the HSAT problem as a hybrid version of the SAT problem which considers linear arithmetic constraints together with boolean SAT constraints. Researchers in [36] present an algorithm to solve the HSAT problem which combines a SAT solving technique [38] with a traditional linear program solver. The algorithm progressively selects variables and explores value assignments while maintaining consistency between the boolean and the arithmetic domains. Other researchers have solved the problem by expressing all constraints in a single domain and using a solver for that domain. In [39] researchers formulate boolean SAT constraints as integer linear arithmetic constraints.

Constraint logic programming (CLP) techniques [40] have been employed which can handle a broad range of constraints including non-linear constraints on both boolean and arithmetic variables. 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 a variety of constraint description formats which allow complex constraints to be captured. Researchers in [41] use the GNUProlog engine [42] to generate tests by converting boolean and arithmetic constraints into Prolog predicates. CLP has also been used to generate tests for path coverage in a control-dataflow graph in [8] where the arithmetic constraints expressed at each branch

point of a path are solved together to generate a test which traverses the path. In [9] researchers employ an approach similar to the one used in [8] to explore a subset of paths which are linearly independent. In [43] the CLP approach is used to generate tests related to the synchronization between concurrent hardware-software processes.

Although Binary Decision Diagrams (BDDs) represent boolean functions by their nature, BDDs have been used at the behavioral level to describe the CDFG of a behavioral VHDL description [44]. These approaches describe arithmetic functions in the boolean domain by describing each output bit function as a BDD. Test patterns are identified by solving SAT for the machine which is the exclusive OR of the good and faulty machines.

State machine testing has been accomplished by defining a *transition tour* which is a path which traverses each state machine transition at least once [17]. Transition tours have been generated by iteratively improving an existing partial tour by concatenating on to it the shortest path to an uncovered transition [17]. In [18], a test sequence is generated for each transition by asserting that a given transition does not exist in a state machine model, and then using the model checking tool SMV [45] to disprove the assertion. A byproduct of disproving the assertion is a counterexample which is a test sequence which includes the transition.

If a fault effect can be observed directly at the machine outputs, then covering each state and transition during test is sufficient to observe the fault. In general, a fault effect may cause the machine to be in an incorrect state which cannot be immediately observed at the outputs. In this case, a *distinguishing sequence* must be applied to differentiate each state from all other states based on output values. The testing problems associated with state machines, including the identification of distinguishing, synchronizing, and homing sequences, are well understood [19].

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

3.2 Coverage Directed Techniques

Several techniques have been developed which generate test sequences without targeting any specific fault. Coverage is improved by modifying an existing test sequence, and then evaluating the coverage of the new sequence. These techniques differ in the method used to modify the test sequence, the cost function used to evaluate a sequence, and the criteria used to accept a new sequence. The modification method is typically either random or directed random.

An example of such a technique is presented in [7] which uses a genetic algorithm to successively improve the population of test sequences. 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 which describe the test sequences, but the mating process defines and restricts the way in which two test sequences are merged. The cost function (or *fitness* function) used to evaluate a test sequence is the total number of elementary operations (variable read/write) which are executed. Work presented in [48] uses a Random Mutation Hill Climber (RMHC) algorithm which randomly modifies a test sequence to

improve a testability cost function. The test sequence modification is completely random and the criteria for accepting a new sequence is that the cost function is improved. The fault model targeted using this approach is the single stuck-at fault model applied to the individual bits of each variable in the behavioral description. The cost function used contains two parts, (1) the number of statements executed by the sequence, and (2) the number of outputs which contain a fault effect.

In [49] researchers generate directed-random pattern sequences to be used for test. No particular fault model is assumed in this approach, so it is up to the user to provide the directives for pattern generation. Two types of directives are used, (1) *constraints* which define the boundaries of the space of feasible test patterns, and (2) *biases* which direct assignments of values to signals in a non-random way. It is the task of the test engineer to develop a set of constraints and biases which will reveal a particular class of faults.

4 Conclusions and Future Directions

We have presented a topology of research efforts in test generation and fault modeling for hardware-software covalidation. It is clear that the field is maturing as researchers have begun to identify and agree on the essential problems to be solved. Our understanding of covalidation has developed to the point that industrial tools are available which provide practical solutions to test generation, particularly at the state machine level. Although automation tools are available, they are not fully trusted by designers and as a result, a significant amount of manual test generation is required for the vast majority of design projects. By examining the state of previous work we can identify areas which should be studied in future work in order to increase the industrial acceptance of covalidation techniques.

A significant obstacle to the widespread acceptance of available techniques is the lack of faith in the correlation between covalidation fault models and real design errors. Automatic test generation techniques have been presented which are applicable to large scale designs, but until the underlying fault models are accepted, the techniques will not be applied in practice. Fault models must be evaluated by identifying a correlation between fault coverage and detection of real design errors. Essential to this evaluation is the compilation of design errors produced by real designers. Research has begun in this direction [50] and should be used to evaluate existing covalidation fault models. Once covalidation fault models are empirically evaluated we can expect to see large increases in covalidation productivity through the automation of test generation.

A great deal of research in hardware-software covalidation is extended from 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 which can result in errors. Such communication requires the use of some asynchronous communication protocol which must be implemented in hardware and in software. Asynchronous communication is a difficult concept for both hardware and software designers, so it can be expected to result in numerous design errors. Hardware-software communication complexity is also increased because inter-processor communication is handled very differently in hardware as compared to software. Hardware description languages typically provide only the most basic synchronization mechanisms, such as the *wait* expression in VHDL. More complicated protocols must be implemented manually and are therefore vulnerable to design errors. Inter-process communication in software tends to use high-level communication primitives such as monitors (i.e. the *synchronized* statement in Java). Although the implementation of each primitive may be known to be correct, the primitive itself may be used incorrectly by the designer,

resulting in design errors. Relatively little research has investigated testing the interfaces between hardware and software components, but this research area is essential.

References

[1] D. D. Gajski and F. Vahid, "Specification and design of embedded hardware-software systems", *IEEE Design and Test of Computers*, vol. 12, pp. 53–67, 1995.

[2] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs", *IEEE Design and Test of Computers*, vol. 18, pp. 36–45, July/August 2001.

[3] B. Beizer, *Software Testing Techniques, Second Edition*, Van Nostrand Reinhold, 1990.

[4] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing", *Software Practice and Engineering*, vol. 21, pp. 685–718, 1991.

[5] G. Al Hayek and C. Robach, "From specification validation to hardware testing: A unified method", *in International Test Conference*, pp. 885–893, October 1996.

[6] A. von Mayrhauser, T. Chen, J. Kok, C. Anderson, A. Read, and A. Hajjar, "On choosing test criteria for behavioral level harware design verification", *in High Level Design Validation and Test Workshop*, pp. 124–130, 2000.

[7] F. Corno, M. Sonze Reorda, G. Squillero, A. Manzone, and A. Pincetti, "Automatic test bench generation for validation of RT-level descriptions: an industrial experience", *in Design Automation and Test in Europe*, pp. 385–389, 2000.

[8] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming", *IEEE Transactions on Very Large Scale Intergration Systems*, vol. 3, pp. 201–214, 1995.

[9] C. Paoli, M.-L. Nivet, and J.-F. Santucci, "Use of constraint solving in order to generate test vectors for behavioral validation", *in High Level Design Validation and Test Workshop*, pp. 15–20, 2000.

[10] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308–320, December 1976.

[11] J. Laski and B. Korel, "A data flow oriented program testing strategy", *IEEE Trans. on Software Engineering*, vol. SE-9, pp. 33–43, 1983.

[12] Q. Zhang and I. G. Harris, "A data flow fault coverage metric for validation of behavioral hdl descriptions", *in International Conference on Computer-Aided Design*, November 2000.

[13] L. White and E. Cohen, "A domain strategy for computer program testing", *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 247–247, 1980.

[14] Q. Zhang and I. G. Harris, “A domain coverage metric for the validation of behavioral vhdl descriptions”, *in International Test Conference*, October 2000.

[15] F. Fallah, S. Devadas, and K. Keutzer, “Occom: Efficient computation of observability-based code coverage metrics for functional verification”, *in Design Automation Conference*, pp. 152–157, June 1998.

[16] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, “Validation vector grade (VVG): A new coverage metric for validation and test”, *in VLSI Test Symposium*, pp. 182–188, 1999.

[17] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, “Architecture validation for processors”, *in International Symposium on Computer Architecture*, pp. 404–413, 1995.

[18] D. Geist, M. Farkas, A. Landver, S. Ur, and Y. Wolfsthal, “Coverage-directed test generation using symbolic techniques”, *in Formal Methods in Computer-Aided Design*, pp. 143–158, November 1996.

[19] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines - a survey”, *IEEE Transactions on Computers*, vol. 84, pp. 1090–1123, August 1996.

[20] K.-T. Cheng and A. S. Krishnakumar, “Automatic functional test bench generation using the extended finite state machine model”, *in Design Automation Conference*, pp. 1–6, 1993.

[21] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, “Abstraction techniques for validation coverage analysis and test generation”, *IEEE Transactions on Computers*, vol. 47, pp. 2–14, January 1998.

[22] J. P. Bergmann and M. A. Horowitz, “Improving coverage analysis and test generation for large designs”, *in International Conference on Computer-Aided Design*, pp. 580–583, 1999.

[23] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Kluwer Academic Publishers, 2000.

[24] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, “Register-transfer level fault modeling and test evaluation techniques for vlsi circuits”, *in Internation Test Conference*, pp. 940–949, 2000.

[25] A. Fin, F. Fummi, and M. Signoretto, “SystemC: A homogenous environment to test embedded systems”, *in International Workshop on Hardware/Software Codesign (CODES)*, 2001.

[26] M. Puig-Medina, G. Ezer, and P. Konas, “Verification of configurable processor cores”, *in Design Automation Conference*, pp. 426–431, June 2000.

[27] J. Shen and J. A. Abraham, “Native mode functional test generation for processors with applications to self test and design validation”, *in International Test Conference*, pp. 990–999, October 1998.

[28] D. Brahme and J. A. Abraham, “Functional testing of microprocessors”, *IEEE Transactions on Computers*, pp. 475–485, June 1984.

[29] A. J. van de Goor and Th. J. W. Verhallen, "Functional testing of current microprocessors", *in International Test Conference*, pp. 684–695, September 1992.

[30] P. Mishra, N. Dutt, and A. Nicolau, "Automatic verification of pipeline specifications", *in High-Level Design Validation and Test Workshop*, pp. 9–13, 2001.

[31] N. Utamaphetai, R. D. Blanton, and J. P. Shen, "Relating buffer-oriented microarchitecture validation to high-level pipeline functionality", *in High-Level Design Validation and Test Workshop*, pp. 3–8, 2001.

[32] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User defined coverage - a tool supported methodology for design verification", *in Design Automation Conference*, pp. 158–163, June 1998.

[33] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design", *in Design Automation Conference*, pp. 178–183, June 1997.

[34] D. Panigrahi, C. N. Taylor, and S. Dey, "Interface based hardware/software validation of a system-on-chip", *in High Level Design Validation and Test Workshop*, pp. 53–58, 2000.

[35] Q. Zhang and I. G. Harris, "A validation fault model for timing-induced functional errors", *in International Test Conference*, October 2001.

[36] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linear programming and 3-satisfiability", *in Design Automation Conference*, pp. 528–533, June 1998.

[37] F. Fallah, A. Pranav, and S. Devadas, "Simulation vector generation from hdl descriptions for observability-enhanced statement coverage", *in Design Automation Conference*, pp. 666–671, June 1999.

[38] T. Larrabee, "Test pattern generation using boolean satisfiability", *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 4–15, January 1992.

[39] Z. Zeng, P. Kalla, and M. Ciesielski, "Lpsat: A unified approach to rtl satisfiability", *in Design, Automation and Test in Europe Conference*, 2000.

[40] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[41] Zeng Z., Ciesielski M., and Rouzeyere B., "Functional test generation using constraint logic programming", *in VLSI-SOC Conference*, 2001.

[42] D. Diaz, *GNU Prolog: A Native Prolog Compiler with Constraint Solving over Finite Domains*, The GNU Project, www.gnu.org, 1999.

[43] F. Xin and I. G. Harris, "Test generation for hardware-software covalidation using non-linear programming", *in High-Level Design Validation and Test Workshop*, 2002.

[44] F. Ferrandi, G. Ferrara, D. Sciuto, A. Fin, and F. Fummi, “Functional test generation for behaviorally sequential models”, *in Design Automation and Test in Europe*, pp. 403–410, March 2001.

[45] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[46] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Implicit state enumeration of finite state machines using BDD’s”, *in International Conference on Computer-Aided Design*, pp. 130–133, November 1990.

[47] S. Sheng and M. S. Hsiao, “Efficient preimage computation using a novel success-driven atpg”, *in Design Automation and Test in Europe*, pp. 822–827, March 2003.

[48] M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, and M. Violante, “Behavioral-level test vector generation for system-on-chip designs”, *in High Level Design Validation and Test Workshop*, pp. 21–26, 2000.

[49] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, “Modeling design constraints and biasing in simulation using BDDs”, *in International Conference on Computer-Aided Design*, pp. 584–589, 1999.

[50] S. Tasiran and K. Keutzer, “Coverage metrics for functional validation of hardware designs”, *IEEE Design and Test of Computers*, vol. 17, pp. 51–60, October-December 2000.