Hardware-Software Covalidation: Fault Models and Test Generation

Ian G. Harris
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
harris@ecs.umass.edu

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 paper presents a summary of research in *hardware-software covalidation* which involves the verification of design correctness using simulation-based techniques. This paper focuses on the test generation process for hardware-software systems as well as the fault models and fault coverage analysis techniques which support test generation.

1 Introduction

The widespread use of hardware-software 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. One issue is the high complexity of hardware-software systems which derives from both the size and the heterogeneous nature of the designs. Hardware verification complexity 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.

In this survey we summarize research in the stages of covalidation involved with test generation. We describe fault models used to describe design defects in Section 2, as well as the automatic test generation techniques which are based on those fault models in Section 3.

2 Fault Models and Coverage Evaluation

A *design defect* is a incorrect feature of a design which is accidentally included by the designer. Design defects may range from simple syntactical 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 defects 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 defects, allowing a large set of design defects 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 allows the concise representation of the set of all design defects for an arbitrary design. Covalidation fault models can be evaluated by their *accuracy* in terms of modeling design defects, and their *efficiency* in terms of the number of faults in a 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].

As a tool to describe covalidation fault models we will use the simple system example shown in Figure 1. Figure 1a shows a simple 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 most 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 model is very efficient since the number of potential faults is equal to the number of lines of code. On the other hand, it is well accepted

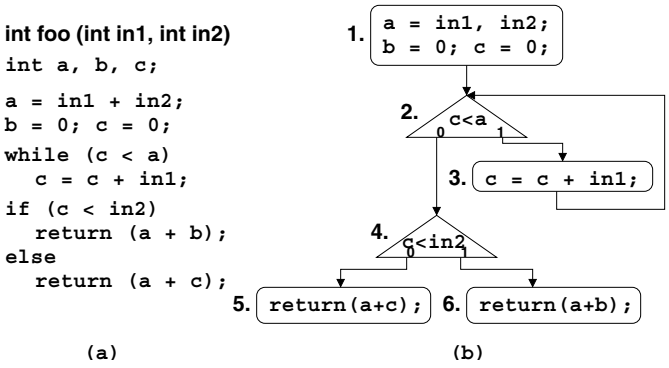

Figure 1: Behavioral Descriptions, (a) Textual Description, (b) Control-Dataflow Graph (CDFG)

that the limited accuracy of statement coverage requires that it be used in conjunction with other fault models in order to properly validate a design.

Mutation analysis is a textual fault model which was originally developed in the field of software test [4, 5], but has also been applied to hardware validation [6]. In mutation analysis terminology, 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 set of mutation operators describes all expected design errors, and therefore defines the functional defect model. Since behavioral hardware descriptions share many features in common with procedural software programs, previous researchers [6] have used a subset of the software mutation operations presented in [4]. A typical mutation operation is *Arithmetic Operator Replacement (AOR)*, which replaces each arithmetic operator with another operator. For example in Figure 1a, the line $a = in1 + in2;$ would be replaced with $a = in1 - in2;$, $a = in1 * in2;$, and $a = in1/in2;$. The efficiency of this metric is good because the number of mutants in a description is $O(s * m)$, where s is the size of the behavioral description and m is the number of mutation operations used, which is a low constant (22 in the case of [4]). The accuracy of this approach has not been demonstrated. The local nature of the mutation operations may limit its ability to describe a large set of design defects.

2.2 Control-Dataflow Fault Models

A number of covalidation fault models are based on the traversal of paths through the CDFG representing to the system behavior. The earliest control-dataflow fault models include 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. Branch coverage is commonly used in for hardware validation and software testing, but it is also accepted to be insufficient to guarantee correctness alone. The efficiency of the branch coverage metric is high because it can be computed by analyzing a single cosimulation output trace. The branch coverage metric has been used for behavioral validation by several researchers for coverage evaluation and test generation [7, 8, 9]. The accuracy of branch coverage has been studied to determine its ability to cover design defects [7, 8]. In [7] researchers found that branch coverage, together with toggle coverage, was sufficient to ensure the detection of 25 of 26 total design defects in a 5-stage pipelined microprocessor example.

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 a defect is associated some path through the control flow graph and therefore all control paths must be executed 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 [10]. 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 [11] (based on work in software test [12]) identifies *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 [13, 14, 15, 16, 17] have 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 [18].

The majority of control-dataflow fault models consider the control-flow paths traversed without over-constraining the values of variables and signals. For example in Figure 1b, in order to traverse path 1, 2, 3, the value of c must be minimally constrained to be less than a, but no additional constraints are required. This can be contrasted with variable/signal-oriented fault models which place more stringent constraints on signal values to ensure fault detection. The domain analysis technique in software test [19, 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. Domain faults may be stimulated by test points anywhere in the input space, but they are most likely to be stimulated by inputs which

cause the program to be in a state which is "near" a domain boundary. An example of this property can be seen in Figure 1b in the traversal of path 1, 2, 3. The only constraint required is that $c < a$, but if the difference between c and a is small, then there is a greater likelihood that a small change in the value of c will cause the incorrect path to be traversed. Researchers have applied this idea to develop a domain coverage fault model which can be applied to hardware and software descriptions [20].

Many control-dataflow fault models consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral fault models [21, 22, 23, 24] to alleviate this weakness. The OCCOM fault model has been applied for hardware validation [21, 22] and for software validation [23]. The OCCOM approach inserts faults called *tags* at each variable assignment which represent a positive or negative offset from the correct signal value. The sign of the error is known but the magnitude is not. Observability analysis along a control-flow path is done probabilistically by using the algebraic properties of the operations along the path and simulation data. A tag will propagate through a behavioral operation if two conditions are met: 1) it is of appropriate sign and, 2) the other inputs to the operation are non-controlling. As an example, in Figure 1 we will assume that a positive tag is inserted on the value of variable c and we must determine if the tag is propagated through the condition $c < in2$ in node 4 of Figure 1b. The propagation of the tag depends on the magnitudes of c and $in2$. Since the tag is positive, it is possible that the conditional statement will execute incorrectly in the presence of the tag, so the OCCOM approach optimistically assumes tag propagation in this case. Notice that a negative tag could not affect the execution of the conditional statement.

While the approach presented in [21, 22, 23] determines observability in a probabilistic fashion, other researchers have developed a a precise technique [24]. Work in [24] injects stuck-at faults on internal variables and determines fault effect propagation behaviorally. Because the observability analysis is precise, the computational complexity is increased.

2.3 State Machine Fault Models

Finite state machines 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 [25, 26, 27] are the *state coverage* model which requires that all states be reached, and *transition coverage* which requires that all transitions be traversed. These fault models have also been refined to differentiate faults in the output function from faults in the next state function [28]. State machine *transition tours*, paths covering each transition of the machine, are applied to microprocessor validation [29]. A user-refined transition coverage model has been proposed [30] which selects only transitions which affect state variables which are identified by the user as being important for test. The problems associated with state machine testing are understood from classical switching theory [31] and are summarized in an thorough survey of on state machine testing [32].

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) [33] and the Extracted Control Flow Machine (ECFM) [26] models create a reduced state machine by partitioning the state bits between control and data bits. In [34] 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. These state machine reduction techniques have successfully enabled validation to be performed for several large-scale designs.

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 [35, 36]. For example, the stuck-at fault model assumes that each signal may be held to a constant value of 0 or 1 due to a defect. The stuck-at fault model has also been applied at the behavioral level for manufacturing test [37] and for hardware-software covalidation [38, 39]. Behavioral designs often use variables which are represented with many bits, and gate-level fault models are typically applied to each bit, individually. For example, if we assume that an integer as declared in Figure 1a is 32 bits long, then applying the single stuck-at fault model to a variable would produce 32 stuck-at-1 faults and 32 stuck-at-0 faults. The toggle coverage fault model, which requires that each bit signal transition up and down, has been applied for design validation and has been expanded to consider observability [24].

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 [40, 41, 42, 43, 44]. Application-specific fault models are also used for popular, well standardized applications such as MPEG coding [45].

Another alternative to a 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. These approaches differ in the method by which the designer specifies the fault model. 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 [46, 47].

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 [48]. 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 [49]. Researchers classify the interface fault which occurred during the design process into three groups: 1) COMP2COMP faults involving communication between pairs of components, 2) COMP2COMM faults involving the interaction between each component and the communication architecture, and 3) COMM faults involving the coordinated interactions between the communication architecture and all components. In [49], test benches are developed manually to target each of these interface fault classes.

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, making them particularly vulnerable to timing-induced faults. Timing-induced faults are described in [50] as faults which cause the definition of a signal value to occur earlier or later than expected. An example of the occurrence of this type of fault would be an increased delay on the *empty* status signal of a FIFO. If the *empty* signal is issued too late, the FIFO may be read from while it is empty. In [20] a timing fault model is presented and a technique for fault coverage evaluation is introduced.

3 Automatic Test Generation Techniques

Several automatic test generation (ATG) approaches have been developed to which vary in the search space technique used, the fault model assumed, the search space technique used, and the design abstraction level used. Our discussion will partition ATG algorithms as either *Fault Directed* techniques which target faults individually, and *Randomized* techniques which target no specific fault but increase fault coverage overall.

3.1 Fault Directed Techniques

State machine testing involves the application test sequences to traverse paths through the machine. Paths are identified through the state machine which include the states and transitions of interest for testing. This goal is sometimes accomplished by defining *transition tours* which are paths containing a subset of all transitions in the state machine [29]. In [30], 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 [51] 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 [32, 31].

The abstraction method used to represent the state machine has been shown to greatly impact the complexity of the test generation process. Binary Decision Diagrams (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 [52]. The efficiency of this method of state enumeration has led to its use during the state machine test generation process [26, 34].

BDDs are also used at the behavioral level to describe the CDFG of a behavioral VHDL description [53, 39, 54]. In these approaches, the functions implemented by each output bit are described as a set of BDDs. Stuck-at faults are inserted at each variable bit to generate faulty BDDs as well. Test patterns are identified by satisfying the machine which is the XOR of the good and faulty machines.

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 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. 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 [55, 56] researchers define the HSAT problem as a hybrid version of the SAT problem which considers linear arithmetic constraints together with boolean 2-SAT and 3-

SAT constraints. Researchers in [55] present an algorithm to solve the HSAT problem which progressively selects variables and explores value assignments while maintaining consistency between the boolean and the arithmetic constraints. Other researchers have solved the problem by using publicly available logic program solving engines such as the CLP(R) engine [57] used in [10] and the GNUProlog engine [58] used in [11].

3.2 Randomized Techniques

Several techniques have been developed which develop test sequences using randomized algorithms to improve overall coverage without a strongly directed search mechanism. An example of such a technique is presented in [59, 9] which uses a genetic algorithm to successively improve the population of test sequences. 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. The goal of this approach is to produce a test sequence which executes each elementary operation at least a minimum number of times. Work presented in [60] uses a Random Mutation Hill Climber (RMHC) algorithm which randomly modifies a test

sequence to improve a testability cost function. 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 [61] researchers use random patterns which are biased by user-defined constraints which alter signal likelihoods based on state conditions.

4 Conclusions

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. The chief obstacle to the widespread acceptance of available techniques is the lack of faith in the correlation between covalidation fault models and real design defects. 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. Once this problem is addressed as part of a growing research effort in hardware-software covalidation, we can expect to see large increases in covalidation productivity through the automation of test generation.

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. Dey, A. Raghunathan, and K. D. Wagner, "Design for testability techniques at the behavioral and register-transfer level", *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 13, pp. 79–91, October 1998.

[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] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators", *ACM Transactions on Software Engineering Methodology*, vol. 5, pp. 99–118, April 1996.

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

[7] 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.

[8] A. Hajjar, T. Chen, and A. von Mayrhauser, "On statistical behavior of branch coverage in testing behavioral vhdl models", *in High Level Design Validation and Test Workshop*, pp. 89–94, 2000.

[9] 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.

[10] 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.

[11] 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.

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

[13] S Rapps and E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Trans. on Software Engineering*, vol. SE-11, pp. 367–375, April 1985.

[14] P. G. Frankl and J. E. Weyuker, "An applicable family of data flow testing criteria", *IEEE Trans. on Software Engineering*, vol. SE-14, pp. 1483–1498, Oct. 1988.

[15] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria", *IEEE Trans. on Software Engineering*, vol. SE-15, pp. 1318–1332, 1989.

[16] S. C. Ntafos, "A comparison of some structural testing strategies", *IEEE Trans. on Software Engineering*, vol. SE-14, pp. 868–874, 1988.

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

[18] 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.

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

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

[21] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation", *in International Conference on Computer-Aided Design*, pp. 418–425, November 1996.

[22] 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.

[23] J. C. Costa, S. Devadas, and J. C. Montiero, "Observability analysis of embedded software for coverage-directed validation", *in International Conference on Computer-Aided Design*, pp. 27–32, November 2000.

[24] 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.

[25] K.-T. Cheng and J.-Y. Jou, "A functional fault model for sequential machines", *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 1065–1073, September 1992.

[26] 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.

[27] N. Malik, S. Roberts, A. Pita, and R. Dobson, "Automaton: an autonomous coverage-based multiprocessor system verification environment", *in IEEE International Workshop on Rapid System Prototyping*, pp. 168–172, June 1997.

[28] A. Gupta, S. Malik, and P. Ashar, "Toward formalizing a validation methodology using simulation coverage", *in Design Automation Conference*, pp. 740–745, June 1997.

[29] 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.

[30] 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.

[31] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw Hill, 1978.

[32] 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.

[33] 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.

[34] 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.

[35] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

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

[37] 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.

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

[39] F. Ferrandi, F. Fummi, L. Gerli, and D. Sciuto, "Symbolic functional vector generation for vhdl specifications", *in Design Automation and Test in Europe*, pp. 442–446, 1999.

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

[41] 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.

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

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

[44] J. Levitt and K. Olukotun, "Verifying correct pipeline implementation for microprocessors", *in International Conference on Computer-Aided Design*, pp. 162–169, 1997.

[45] P. Meehan, N. Hurst, M. Isnardi, and P. Shah, "Mpeg compliance bitstream design", *in International Conference on Consumer Electronics*, pp. 174–175, June 1995.

[46] 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.

[47] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosentiel, "Checking temporal properties under simulation of executable system descriptions", *in High Level Design Validation and Test Workshop*, pp. 161–166, 2000.

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

[49] 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.

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

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

[52] 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.

[53] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral vhdl models", *in International Test Conference*, pp. 587–596, October 1998.

[54] 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.

[55] 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.

[56] 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.

[57] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap, "The clp(r) language and system", *ACM Transactions on Programming Languages and Systems*, vol. 14, pp. 339–395, July 1992.

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

[59] F. Corno, P. Prinetto, and M. Sonza Reorda, "Testability analysis and atpg on behavioral rt-level vhdl", *in International Test Conference*, pp. 753–759, 1997.

[60] 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.

[61] 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.