Test Pattern Generation for Timing-Induced Functional Errors in Hardware-Software Systems

Srikanth Arekapudi, Fei Xin, Jinzheng Peng and Ian G. Harris
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
E-mail: {sarekapu,fxin,jpeng,harris}@ecs.umass.edu

Abstract

We present an ATPG algorithm for the covalidation of hardware-software systems. Specifically, we target the detection of timing-induced functional errors in the design by using a design fault model which we propose. The computational time required by the test generation process is sufficiently low that the ATPG tool can be used by a designer to achieve a significant reduction in validation cost.

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. 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, we are investigating *covalidation* techniques, in which functionality is verified by simulating (or emulating) a system description with a given test input sequence. In contrast, 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 complexity of formal techniques make covalidation the only practical solution for many designs.

Previous work in validation has concentrated on *unit testing*, the validation of a single task or process. These testing approaches identify static errors, those errors which directly impact data values, independent of the time between the application of test datum. Hardware-software systems are susceptible to timing errors which directly impact the time of the application of data rather than the value of that data. The main difference between the detection of timing and static errors is the duration of an erroneous data state. A timing error may cause a signal to have an incorrect value for a short time period which cannot be controlled by manipulating the test sequence. The notion of validating timing constraints has not been adequately addressed in either the software or the hardware domains. Modeling timing constraints during validation is central to the hardware-software covalidation problem. Existing software testing models must be enhanced to include timing constraints, and to model timing related defects.

The covalidation process typically requires a time-consuming manual test generation step. We propose an automatic test pattern generation (ATPG) tool which can be used to greatly reduce the time required for covalidation. The result of the ATPG process is a timed sequence of events on the system inputs which will detect timing-induced faults described by our design fault model. The ATPG algorithm uses a Co-design Finite State Machine (CFSM) model [1] to capture the system behavior, and to express the interactions between system components. The CFSM model has the advantage that it is supported by the POLIS co-design framework [2], and it can be constructed directly from reactive languages including ESTEREL [3].

The paper is organized as follows: Previous work in hardware-software covalidation is presented in Section 2. Section 3 describes proposed design fault model for timing-induced errors. Section 4 outlines the stages of a test pattern generation technique to target the proposed timing fault model. Results are presented in Section 5 and the broader impacts of the work are summarized in Section 6.

2. Previous Work

A survey outlining fault models and test generation for hardware-software covalidation is presented in [4]. Covalidation fault models have been developed at different levels of abstraction, each model defining a set

of expected design defects. Fault models have been developed directly at the behavioral level in [5] and [6] where a fault model assumes that any single variable assignment in a behavioral description may be incorrect. In [7], the authors use the fault model presented in [6] to build a test generation tool based on the 3-Satisfiability problem. Mutation analysis has been used for hardware validation previously in [8] by converting a VHDL program into a functionally equivalent Fortran program and then using the Mothra tool for software mutation analysis [9]. Both domain testing and dataflow testing methods have been previously applied to the validation of behavioral VHDL descriptions [10, 11]. Although previous techniques have analyzed system performance at the task level [12] without considering functional errors, we have considered the detection of timing-induced functional errors in [13].

3. Modeling Real-Time Design Errors

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 *design fault model* describes the definition of a set of faults for an arbitrary design. A design fault model allows the concise representation of the set of all design defects for an arbitrary design.

Several design fault models have been proposed previously in the area of software testing, in the context of *dataflow analysis* testing. We have modified existing dataflow analysis techniques to capture timing-induced functional errors [13]. A timing fault exists when a signal is assigned to the correct value, but the assignment event occurs at the incorrect time. A timing fault will cause a signal value to endure for the incorrect length of time. The timing fault effect can be observed only during the incorrect time period. The difference between static faults and timing-induced faults is that a timing fault is active during only a subset of the time period between two definitions, while a static fault is active during the entire time period between two definitions.

To describe the detection properties of timing faults, we will use a small system example in which Process X sends data to Process Y through a FIFO buffer. To explain the fault model we must describe some terminology. Each signal occurrence in a behavioral description is classified as either a *definition occurrence* or a *use occurrence*. A definition occurrence describes a statement where a value is bound to a signal. A use occurrence describes a statement which refers to the value of a signal.

There are several signal timing relationships which must be maintained to guarantee correct communication between the two processes. Typical timing constraints for FIFO-based communication include the maximum latency on output signals such as the *empty* signal. If the empty signal is asserted later than expected, then Process Y may attempt to read data from an empty buffer. Figure 1 depicts the timing details involved with a late *empty* signal. Figure 1a shows the definition of the *empty* signal in the FIFO description where *empty* signal is asserted. Before Process Y can read data from the FIFO, it must check the *empty* signal as shown in Figure 1b. The event trace shown in Figure 1c shows both the correct and the late assertion times of the *empty* signal. The highlighted region which is referred to as the *error span* is the time during which the *empty* signal has the incorrect value. If there is a use occurrence during the error span, then that use will receive different data values in the correct and the faulty circuits, and the fault can be detected.

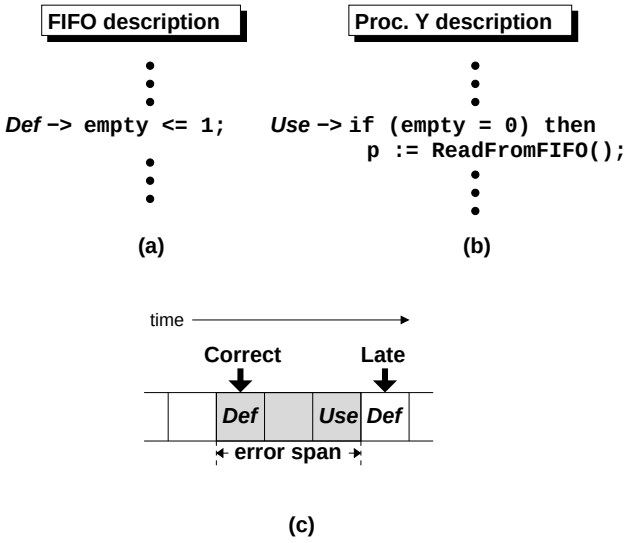

Figure 1: *empty* **signal is asserted late, (a) a section of the FIFO description, (b) a section of the Process Y description, (c) event trace with error span highlighted.**

We define a **Mis-Timed Event (MTE)** fault to be associated with each pair of definition and use statement pairs on a given signal $s \in S$, where S is the set of all signals used in the design. The existence of an MTE fault indicates that the associated signal defini-

tion occurs at the incorrect time and causes the associated use to receive incorrect data. Two types of MTE faults can exist, MTE_{early} where the definition occurs earlier than the correct time, and MTE_{late} where the definition occurs later than the correct time.

3.1. Detection of Timing Faults

The example of Figure 1 demonstrates that a timing fault associated with a signal is detected only if there is a use of the signal inside the error span of the fault. The error span extends from the erroneous time step to the correct time step. Unfortunately, the precise position of the error span is not known since simulation of the faulty circuit reveals only the erroneous time step. It is clear, however, that the error span must extend, either forward or backward in time, from the erroneous time step. In order to ensure that a use occurrence is within the error span of a fault, the use occurrence must be close to the corresponding definition occurrence in time. Also, a use occurrence must exist both earlier than the definition and later than the definition to detect both late and early MTE faults. The detection of the MTE_{late} fault is accomplished by the use *before* the erroneous time step, and the MTE_{early} fault is detected by the use *after* the erroneous time step.

4. Automatic Test Pattern Generation

The goal of automatic test pattern generation is to identify a timed sequence of input patterns which will cause the detection conditions of a given timing fault to be satisfied. Starting from a hardware-software description, a set of faults is initially generated. An undetected fault is then selected, effectively at random. A test sequence is generated for the selected fault and fault simulation is performed to identify all other faults which are also detected by the sequence. This process is repeated until either all faults are detected or have been shown to be undetectable under the given detection parameters. The fault list generation, test sequence generation, and fault simulation are described in the following sections.

4.1. Fault List Generation for CFSMs

In order to identify an input sequence, a model of the system computation is needed which can be systematically evaluated during the test generation process. Since we are investigating timing faults, a model is needed which exposes timing information. We formulate the MTE fault model to be compatible with the Co-design Finite State Machine (CFSM) [1] computational model which allows the passage of time to be represented. The CFSM was defined as a compact and intuitive model for the description of hardware-software systems. A system is described as a network of CFSMs, where each CFSM describes a concurrent process in the system. The CFSMs communicate via events on signals. Each event is identified by a name, a value, and a time of occurrence. Each CFSM in a system contains a set of states and a transition relation which can be described as a set of edges in a graph in which each state is represented by a node. Each edge is a *cause-reaction* pair where the cause is a set of event names and values, and each reaction is a set of events and values. When an edge is triggered by an event which matches its cause, the CFSM changes state to the destination state of the edge, and all events in the reaction set are emitted. There is a nonzero time between the cause and the effect which, in practice, would be determined using some performance estimation technique. The example in Figure 2 shows two CFSMs which model a Process X which feeds data to a FIFO. Each edge in the CFSMs is labeled *cause -> reaction*, unless the edge does not involve a reaction in which case only the cause is shown. CFSMs include

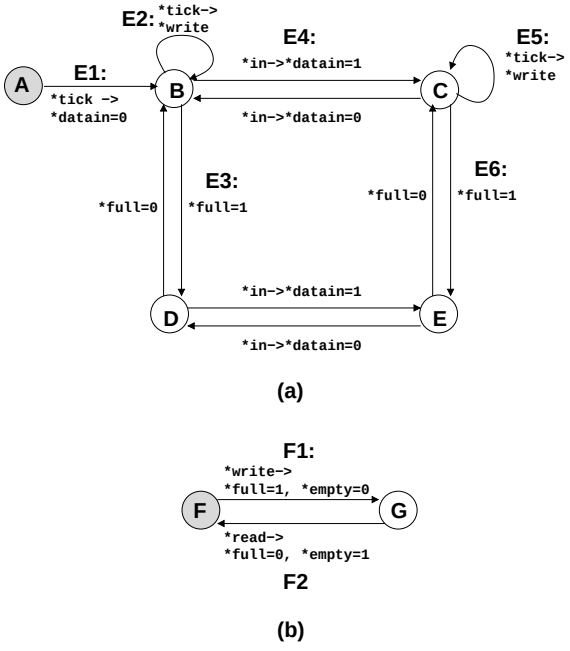

Figure 2: **Co-design Finite State Machine Example, (a) Process X CFSM, (b) FIFO CFSM**

two types of signals, trigger signals and value signals. Trigger signals (denoted with a * prefix) implement the basic synchronization mechanism of a CFSM. Trigger events can be used to cause a transition in a CFSM. This is similar the *sensitivity list* concept in VHDL and other hardware description languages. Value signals may have an arbitrarily large domain and their values persist until the signal value is reassigned. Value

signals cannot cause a transition, but can be used to choose among different possibilities. Each edge in a CFSM must be caused by at least one trigger signal.

In order to apply the proposed fault model to CFSMs, we must identify definition and use statements in a CFSM. Signal definitions exist at each *reaction* associated with an edge because the reactions assign values to signals. Signal uses are the *causes* of each edge because the value of a signal causing a transition must be detected. By this definition, a definition-use pair maps to a pair of edges in the CFSM network; one edge includes the definition as one of its *reactions*, and the other edge includes the use as one of its *causes*.

Timing-induced functional errors associated with value signals are considered. The fault list is determined by matching all definition-use pairs on each value signal which is internal to the system. For each definition-use pair involving the same signal and signal value, an MTE_{late} and an MTE_{early} faults are created because late and early execution of the definition may cause the use to receive the incorrect value.

4.2. Test Sequence Generation

Test pattern generation identifies a sequence of events on input signals which will cause a given definition and use pair to occur within time δ. A test sequence which detects a timing fault on a signal must trigger the system to perform a computation in which the signal is defined and used within a fixed time period δ. Test pattern generation requires the identification of a computation which satisfies the detection requirements of the fault. A *computation* of a system can be defined informally as the sequence of events resulting from a given input sequence. In a CFSM, the event sequence produced is determined by the path through the CFSM which is executed. A computation is referred to as a *timed* computation when all of the edges contained in the computation paths are mapped to time steps. Once the timed computation is identified, the test sequence is determined by the set of events associated with the input signals.

In order to ensure that a solution is found for each non-redundant MTE fault, the ATPG algorithm must be essentially enumerative in the worst case, exploring *all* possible timed computations whose duration is less than some maximum threshold. The requirement for enumeration causes any ATPG algorithm to have non-polynomial time complexity, but by using heuristics we attempt to make the complexity tractable in the average case. The task of identifying a timed computation to detect a timing fault can be subdivided into the following steps:

Path Identification - A path which represents the state transitions of a CFSM is identified in each CFSM. A path in a CFSM is defined as a sequence of edges since all event information in a CFSM is associated with the edges rather than the states. The set of paths selected must satisfy the detection requirements of a fault, and the paths must be *compatible*. Two paths are incompatible in a computation if they cannot both exist in the same computation due to conflicting edge triggering requirements. Figure 3 shows an example of conflicting paths for the CFSMs in Figure 2. Each node in a path of Figure 3 represents an edge of a CFSM in Figure 2. Figure 3a shows two paths which are incompatible because the number of edges in Path 2 which are triggered by the assertion of the $*write$ signal is larger than the number of assertions of the $*write$ signal which occur in Path 1. Path 2 cannot be traversed concurrently with Path 1 because it is not possible to trigger both occurrences of edge F1 with only a single assertion of the $*write$ signal. Figure 3b shows two paths which are compatible because the number of edges in Path 2 which are triggered by the assertion of the $*write$ signal is equal to the number of assertions of the $*write$ signal which occur in Path 1. Our algorithm for path identification is enumerative. All sets of paths whose length is less than a fixed limit are explored successively. Path sets which are not compatible as explained earlier are not explored further.

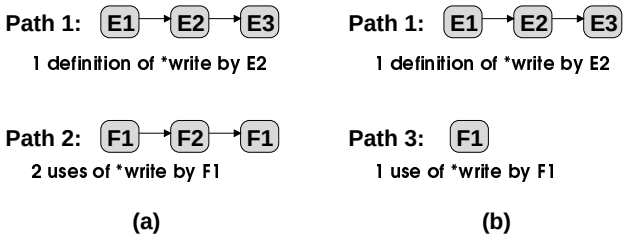

Figure 3: **CFSM Paths, (a) Incompatible Paths, (b) Compatible Paths**

Trigger Event Matching - Each edge in a CFSM path must be matched with an event on a trigger signal which causes the edge to be traversed. This matching is required to ensure that each CFSM traverses the specified paths. An example of this matching is shown with the paths in Figure 4. The bottom row contains the definitions of the $*tick$ signal which are created by the clock and are placed at fixed intervals according to the clock frequency. Two paths are shown, Path 1, E1, E2, E3 and Path 2, F1. Each element of a path is the cause-reaction pair associated with the corresponding CFSM edge. The cause is a signal use which detects an event which triggers edge traversal. The reaction is the set of definition events which occur after the edge is triggered. Each edge cause is matched to a definition event of the same signal. The matching of signal definitions to transition causes is shown in Figure 4 by shading the matching pair. All causes in Figure 4 are matched to definition events. For a given path set,

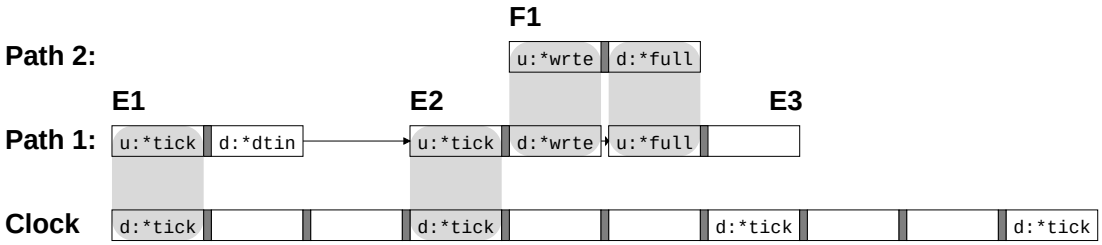

Figure 4: **Trigger event matching, a feasible matching between two paths**

our trigger event matching algorithm enumerates the set of all feasible matchings for further exploration. A matching may be infeasible because a state is sensitive to an unmatched definition, or because a use cannot be matched due to ordering constraints. Matchings which are infeasible for any reason are not explored further.

Timing Resolution - Each event which triggers a CFSM edge must be mapped to a time step. The test sequence is the set of events on input signals, so this step completes the test sequence definition by mapping all input events to time steps. All signal definitions and triggers which are matched during trigger event matching must be mapped to the same time step.

Timing resolution is formulated as a linear program in which the execution time of each edge is represented as a variable. Linear equations are used to express ordering constraints between adjacent path elements. Trigger event matching has the effect of forcing matched events to occur at a fixed time separation. Matching constraints are also expressed as linear equations. The problem may be expressed as a linear program as long as the range of time in which each edge can be scheduled is continuous. If the time range of an edge is not continuous (possibly interrupted by a sensitive state) then a single continuous subrange must be selected for each edge.

4.3. Fault Simulation

The task at this stage is to identify all faults which are detected by a given test sequence and label these faults as being detected. A byproduct of the test sequence generation process is a complete timed computation which contains the time of each event. By examining the timed computation, it is simple to locate all definition-use pairs for which the fault detection criteria outlined in Section 3.1 are satisfied.

5. Experimental Results

In order to evaluate our ATPG tool we have used it to develop tests for a system based on the gas station problem [14]. The gas station problem is a simulation of an automated self-serve gas station. Our version of the gas station consists of three tasks: the Customer, the Server, and the Pump. The Pump can provide discrete amounts of gasoline, either 5, 10, or 15 gallons. When a car arrives, a sensor associated with the $*car$ signal notifies the Station. When the Station detects the car, the Station requests money (via the $*pay$ signal) according to the amount of fuel required. The $paykey$ input is used to indicate the amount of gasoline required. The Customer pays for the fuel (via the pay signal). After payment, the Pump pumps the appropriate amount of fuel and notifies the station on completion. The Station then returns the change via the $*bill$ output and goes to its idle state to await the next car. The CFSMs for both the Station and the Customer tasks and shown in Figure 5. Each edge in the CFSMs is labeled, S1 - S3 in the Station CFSM and C1 - C3 in the Customer CFSM. The $*tick$ signal is the output of a clock. The CFSM of the Pump task is not shown here for brevity since it is not relevant for the detection of the faults we discuss. Figure 6 shows the detailed

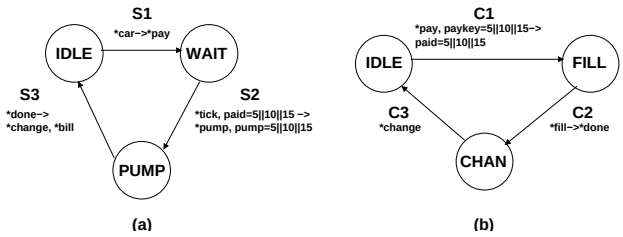

Figure 5: **The Gas Station Problem, (a) Station CFSM, (b) Customer CFSM**

result produced to detect the MTE_{late} fault associated with the definition of $paid = 10$ in edge C1, and the use of $paid = 10$ in edge S2. Two CFSM paths are shown, one containing 2 edges in the Station CFSM, and the other containing 1 edge in the Customer CFSM. Each edge is labeled with the use occurrences which are its $causes$, and the definition occurrences which are its

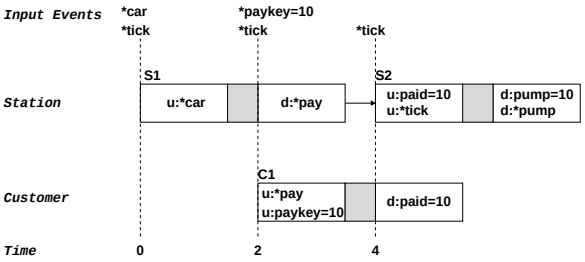

Figure 6: **Test Sequence for MTE faults on** $paid = 10$

delay	δ	FC	CT	PS	MC	LP
1	2	100.00%	0.04s	8	3	3
1	5	100.00%	0.01s	8	3	3
2	2	100.00%	0.02s	8	3	3
2	5	100.00%	0.01s	8	3	3

Table 1: **Gas Station ATPG Results, ML=2, CLK=2, TF=6**

reactions. Each edge is also placed at the point in time where is occurs. In this example a delay of 2 is used, so each edge *reaction* is 2 time steps after its *cause*. The input events which comprise the test sequence are shown along the top row, and are placed in time. The MTE_{late} fault is detected when the $*car$ and $*paykey = 10$ inputs events occur some small ϵ time before time steps 0 and 2 respectively because this forces the definition of $paid = 10$ to occur ϵ time before the use.

We used our ATPG tool to detect each of the MTE faults. Each ATPG run varied on at least one of four parameters: ML - maximum path length, CLK - the period of the clock, δ - the error span limit, and *delay* - the delay associated with each CFSM edge. For simplicity we assume that each edge has the same delay. Table 1 shows the ATPG results over a range of parametric values. The output statistics considered are TF - Total faults, FC - Fault coverage, CT - CPU Time including time required by the linear program solver, PS - Number of feasible path sets explored, MC - Number of feasible trigger event matchings considered, LP - Number of distinct linear programs which are solved.

6. Conclusions

We present an automatic test pattern generation technique for the covalidation of hardware-software systems. The expense of covalidation and the widespread use of hardware-software systems motivates the significance of research in this area. The tractability inherent in simulation-based techniques give covalidation the potential to enjoy acceptance in industry which has not been gained by other verification approaches. By targeting timing faults in this proposal we are investigating an essential class of faults which has not been fully understood in previous work on covalidation.

7. References

[1] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, pp. 26–36, August 1994.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers, 1997.

[3] F. Boussinot and R. deSimone, "The esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, September 1991.

[4] I. Harris, "Hardware-software covalidation: Fault models and test generation," *To appear in IEEE International High Level Design Validation and Test Workshop*, 2001.

[5] F. Fallah, P. Ashar, and S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability Enhanced-Statement Coverage," in *Proceedings of the* 36^{th} *Design Automation Conference*, June 1999, pp. 666–671.

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

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

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

[9] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software Practice and Engineering*, vol. 21, no. 7, pp. 685–718, 1991.

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

[11] 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.

[12] A.Dasdan, D.Ramanathan, and R.K. Gupta, "A timing-driven design and validation methodology for embedded real-time systems," in *ACM Trans. Design Automation of Electronic Systems*, October 1998, vol. 3.

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

[14] D. Helmbold and D. Luckham, "Debugging ada tasking programs," in *IEEE Software*, March 1985, pp. 47–57.