A Validation Fault Model for Timing-Induced Functional Errors

Qiushuang Zhang and Ian G. Harris
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
qzhang@ecs.umass.edu, harris@ecs.umass.edu
Phone: 413-545-1594, Fax: 413-545-1993

Abstract—

The violation of timing constraints on signals within a complex system can create *timing-induced functional errors* which alter the value of output signals. These errors are not detected by traditional functional validation approaches because functional validation does not consider signal timing. Timing-induced functional errors are also not detected by traditional timing analysis approaches because the errors may affect output data values without affecting output signal timing. A timing fault model, the Mis-Timed Event (MTE) fault model, is proposed to model timing-induced functional errors. The MTE fault model formulates timing errors in terms of their effects on the lifespans of the signal values associated with the fault. We use several examples to evaluate the MTE fault model. MTE fault coverage results shows that it efficiently captures an important class of errors which are not targeted by other metrics.

I. Introduction

The widespread use of complex hardware systems in cost-critical and life-critical applications motivates the need for a systematic approach to verify functionality. 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 validation 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. Formal verification techniques have the advantage that they are precise, where validation can only provide a degree of certainty which is less than 100%. However, formal techniques suffer from high complexity, so the verification of large designs using formal techniques alone is often intractable. The complexity of validation can be made tractable by using a test sequence of reasonable length, and the degree of certainty provided can become arbitrarily close to 100%. We investigate validation techniques which can be used in conjunction with formal verification techniques to verify large hardware systems.

A practical difficulty in the validation of large hardware systems is choosing the proper design abstraction level which provides a tradeoff between simulation complexity and error modeling accuracy. In practice, validation is performed at all levels of abstraction from behavioral down to layout. Behavioral hardware description languages, such as VHDL and Verilog, have only been fully accepted by industry for less than a decade, and research in behavioral validation is still developing. Behavioral software languages have been widely used for several decades, so it is to be expected that previous work in software testing may be leveraged to address the validation problem. Several key differences exist between software languages and hardware descriptions languages which must be studied before software testing techniques can be applied. The hardware design process must consider the timing of events inside the system to guarantee correct design. Hardware description languages support time-varying *signals*, and include concurrency constructs such as the *process* statement in VHDL. The notion of validating internal timing activity at the behavioral level has not been adequately addressed in either the software or the hardware domains. Modeling internal timing constraints during validation is central to the hardware validation problem. Existing software testing models must be enhanced to include timing relationships, and to model timing-induced errors.

Previous work in validation test 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. The functionality of a hardware system depends on the correctness of the communication between processes, as well the correctness of each individual process. Since each process may be timed by a different clock, inter-process communication must be properly synchronized in time. Hardware systems are therefore susceptible to internal timing errors which

directly impact the time of the application of data rather than the value of that data. A timing-induced error may cause a signal to have an incorrect value for a short time period which cannot be controlled by manipulating the test sequence. Timing-induced errors may therefore manifest themselves as transient errors whose effects must be detected within a small window of time. The detection requirements for timing-induced errors are not satisfied by conventional functional validation techniques.

Additionally, timing-induced errors may impact output data values without affecting output data timing, and may therefore be ignored by timing analysis. An example of this type of error would be a system described in [25] and shown in Figure 1. The system periodically takes an input from an analog/digital converter, performs a computation on it, and outputs the result to a digital/analog converter. To simplify the example we assume that Computation X produces an output equal to the input. If Computation X completes within a single sample period then the output is a time shifted version of the input. However, if a design error causes the computation to take longer than a sample period, then the data output at each time period will be the incorrect sample, and the output signal will be incorrect. Notice that the timing error does not necessarily effect the rate at which the total system produces output data. The digital/analog converter can produce a new output at each sample period, but the output values will be incorrect. We refer to this error as a timing-induced functional error because it is caused by an internal timing problem, but it manifests itself as a functional error when viewed from outside of the system. A timing-induced functional error will not be detected by task-level timing analysis [6], [5] because the overall system still produces data at the correct rate. To detect these errors a new fault model is required which considers the relationship between timing and functionality.

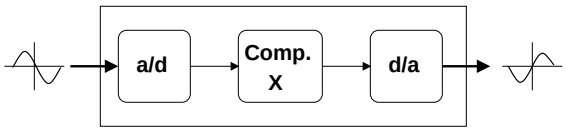

Fig. 1. A system to process an analog signal

The paper is organized as follows: Previous work in hardware validation is presented in Section II. Section III describes the design fault model for timing errors. A fault simulation method is presented in Section IV. Results are presented in Section V and Section VI presents conclusions.

II. Previous Work

A. Hardware Validation

Fault models have been developed at different levels of abstraction, each model defining a set of expected defects. Logic level models [16], [1] assume defects such as the use of an incorrect gate, insertion of an extra line, deletion of a line, and deletion of a gate. In [16], the defect model is used to direct an automatic test generation tool which is presented. A more broad logic level defect model is presented in [15] which considers any defect which can be repaired by re-synthesizing a single signal in the circuit. In [12] a fault model is presented at the finite state machine level which assumes that each error affects either a single state transition or a single transition output.

Fault models have been developed directly at the behavioral level in [8] and [7] where a fault model assumes that any single variable assignment in a behavioral description may be incorrect. This is represented by associating each variable assignment with both a positive and negative tag to represent both assignment incorrect possibilities. The tags are propagated through the control-flow graph using a set of tag propagation rules which consider masking effects. In [9], the authors use the fault model presented in [7] to build a test generation tool based on the 3-Satisfiability problem. Mutation analysis has been used for hardware validation previously in [13] by converting a VHDL program into a functionally equivalent Fortran program and then using the Mothra tool for software mutation analysis [17]. Researchers have applied software path testing to VHDL by allowing the user to select control-flow paths to stimulate, and using constraint programming to identify tests to stimulate the chosen paths [24]. The tool presented in [11] act as a simulator and data collector, allowing the user to specify the nature of the fault coverage to be computed. We have previously applied both domain testing and dataflow testing methods to the validation of behavioral VHDL descriptions [28], [27]. Previous work in timing verification has studied the impact of design errors on timing correctness [3], [19], [26]. Researchers have developed techniques for static timing analysis of hardware-software systems.

B. Software Validation

Software researchers have been studying the problem of validating behavioral descriptions and have developed several techniques which can be applied in hardware validation. The earliest software fault coverage metrics include statement coverage, branch coverage, and path coverage [2]. Statement coverage assumes that the execution of a faulty statement will guarantee the detection of the fault. The branch coverage metric complements statement coverage by reflecting the number of branches

which are taken at some point during testing. The path coverage metric is a more demanding metric than either the statement or branch coverage metrics because path coverage reflects the number of control-flow paths taken. Since the total number of control-flow paths grows exponentially with the number of conditional statements, achieving high path coverage is a highly complex task. Data flow based test adequacy is a structure based test adequacy criteria which is concerned with the occurrences of variables in a program. Each variable occurrence is classified as either a definition occurrence or a use occurrence. The basic criteria [22], [10], [4], [20], [18] identify a subset of paths through the dataflow graph which must be traversed during testing. Mutation analysis [17], [21] is similar to fault simulation using a set of *mutation operations* which describe the expected defects. The number of mutants can be high, making this approach time consuming, but research has been performed to limit the number of mutants [21], and to weaken the mutation detection requirements [14].

III. Modeling Timing Design Errors

A *design error* is a incorrect feature of a design which is accidentally included by the designer. Design errors 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 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 larger set of design errors 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 errors 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. These techniques identify control paths which must be traversed during testing. Several test adequacy criteria based on dataflow analysis have been developed [22], [10], [20], [18]. We propose to modify existing dataflow analysis techniques to capture timing errors. We will first describe the traditional dataflow analysis techniques, and then we will describe the new formulation for timing errors.

Dataflow analysis for HDL descriptions [27] is concerned with the occurrences of variables in a HDL description. Each variable occurrence in a VHDL description is classified as either a definition occurrence or a use occurrence. A *definition occurrence* of a variable describes a statement where a value is bound to the variable. A *use occurrence* of a variable describes a statement which refers to the value of the variable. This occurrence information is added to the CDFG representation as a preprocessing step to facilitate dataflow analysis. Figure 2 shows the CDFG of a simple VHDL description. Note that a node in the graph can have multiple use occurrences of a variable but no more than one definition occurrence of that variable. After the execution of a node completes, the nodes pointed to by outgoing solid edges begin to execute if the condition on the edges are satisfied.

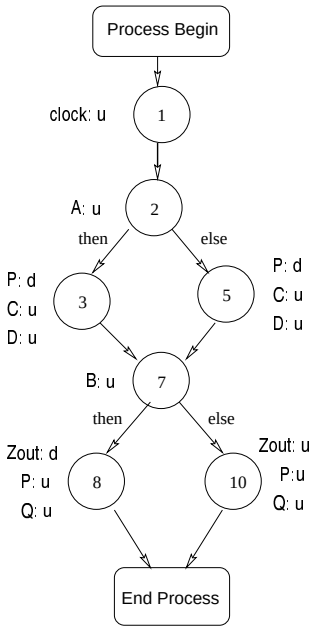

Fig. 2. Flow graph with data flow information.

Based on the flow graph model introduced above, a *definition clear path* with respect to a variable X is a path in the flow graph without definition occurrence of X. A *definition-use (du) pair* of a variable X consists of a definition and a use of variable X which are connected by a definition clear path with respect to X, from the definition to the use. If a *du* pair is exercised in the definition-use sequence by some test patterns, then the *du* pair is covered by the test patterns. *All definition-use (du) pairs metric* [22] requires that all *du* pairs be covered by the test patterns, i.e. every definition to every use of that definition should be exercised. In Figure 2, there are four *du* pairs of variable P, $(3->8)$, $(3->10)$, $(5->8)$ and $(5->10)$, and these *du* pairs are required to be executed by all *du* pairs metric.

A. Timing Fault Model

Design faults can be grouped into two classes, static faults whose observation is independent of absolute event

timing, and timing faults whose observation depends on a specific timing of events on input signals. The observation of a static fault depends on the sequence of test pattern application, but not the absolute time of the application of each pattern. An example of a static fault is the replacement of the expression $x <= y + 1$ with the incorrect expression $x <= y + 2$. Once this fault is activated, its effects can be observed at any time before the signal x is redefined. A timing fault exists when a signal is assigned to the correct value, but the 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 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 the small example shown in Figure 3 in which Process X is sending data to Process Y through a FIFO buffer. The FIFO has 3 inputs, (1) *datain*, which takes input data, (2) *write*, which is asserted when new data is to be written to the FIFO, and (3) *read* which is asserted when data is to be read from the FIFO. The FIFO also has 3 outputs, (1) *dataout* which is driven with output data when a read is performed, (2) *empty* which indicates that the buffer contains no data, and (3) *full* which indicates that no new data can be written to the buffer. In the following examples we assume a discrete event timing model which is commonly used with hardware and hardware-software description languages. Although we assume the discrete event model for explanation purposes, the fault model is not limited in this way and we will investigate the use of different timing assumptions as part of this research.

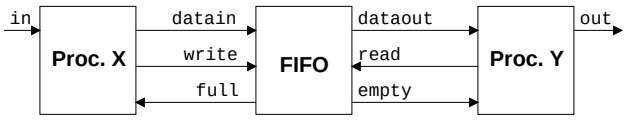

Fig. 3. Two processes communicating via a FIFO

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 4 depicts the timing details involved with a late empty signal. Figure 4a 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 4b. The event trace shown in Figure 4c 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 will be detected.

In addition to events occurring later than expected, events occurring earlier than expected can create incorrect results as well. For example, when Process X writes data to the FIFO the *write* signal must be asserted after the *datain* lines receive the data to be written. If the *write* signal is issued early then it may occur before data is ready on the *datain* lines. The fault is associated with the *du* pair shown in Figures 5a and 5b. The *datain* lines are defined in the code shown in Figure 5a, and the *datain* lines are inserted into the buffer in the code shown in Figure 5b. The event trace in Figure 5c depicts the error span associated with the fault.

We can now define a fault model which describes the set of timing faults potentially contained in a hardware-software description. In order to do so, we must make clear the distinction between a definition (use) *statement* and a definition (use) *occurrence* in our terminology. A *statement* refers to a statement in the original procedural specification of the hardware-software system, while an *occurrence* refers to the execution of a statement during simulation. A single statement may executed many times during simulation, and may therefore be associated with many occurrences.

Definition - A definition occurrence is a tuple $d_o = (d_s, t)$ and a use occurrence is a tuple $u_o = (u_s, t)$:

- $d_s \in D_s$, where D_s is the set of all statements in the hardware-software description which assign a value to signal s.
- $u_s \in U_s$, where U_s is the set of all statements in the hardware-software description which use the value of signal s.
- t is a non-negative integer representing the time of the occurrence.

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 definition 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.

Definition - An MTE_{early} (MTE_{late}) fault is a tuple $m =$

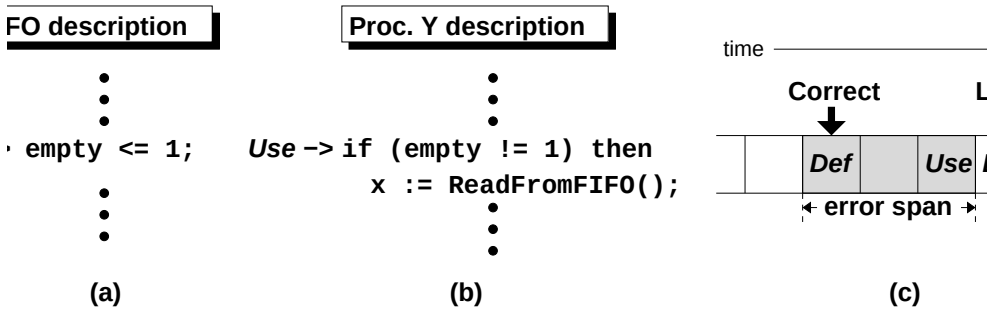

Fig. 4. *empty* signal is asserted late, (a) a section of the FIFO description, (b) a section of the Process Y description, (c) event trace with error span highlighted.

(d_s, u_s).

For example, Figure 4 shows an MTE_{late} fault and Figure 5 shows an MTE_{early} fault.

B. Detection of Timing Faults

The examples of Figures 4 and 5 demonstrate 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

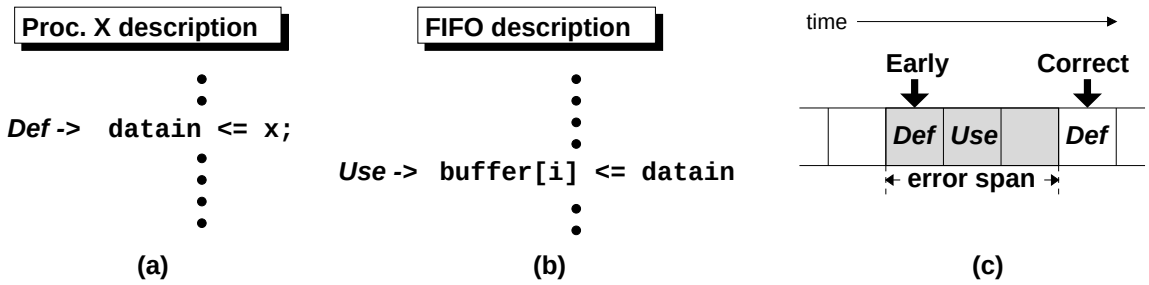

Fig. 5. *datain* signal is asserted early, (a) a section of the Process X description, (b) a section of the FIFO description, (c) event trace with error span highlighted.

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. These circumstances exist in Figures 4c and 5c where, in each case, the use occurrence is immediately adjacent to the erroneous time step. 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. We state these fault detection requirements given a test sequence P as follows.

Definition - An MTE_{early} fault, $m = (d_s, u_s)$, is detected if there exists $(d_s, t_1) \in DO_{s,P}, (u_s, t_2) \in UO_{s,P}$, such that $t_2 - t_1 < \delta$.

Definition - An MTE_{late} fault, $m = (d_s, u_s)$, is detected if there exists $(d_s, t_1) \in DO_{s,P}, (u_s, t_2) \in UO_{s,P}$, such that $t_1 - t_2 < \delta$.

- $DO_{s,P}$ is the set of definition occurrences of signal s during simulation with a test sequence P.
- $UO_{s,P}$ is the set of use occurrences of signal s during simulation with a test sequence P.
- δ is the *error span threshold*, a non-negative integer representing the maximum time between the definition and use occurrence. δ is also the minimum size of an error span which is guaranteed to be detected.

IV. Timing Fault Simulation

We define fault simulation as the process of determining the number of MTE faults detected by simulating the design with a given test sequence. For the fault simulation results shown here we have used the SystemC language [23] which is freely available and allows simulation by compilation to a C++ executable. MTE fault simulation is consists of three steps.

1 . du/ud pairs identification. The detection of MTE faults requires a use immediately before and after the definition. The representation of the requirement on a data flow in a definition-use pair and a use-definition pair. So the first step is to identify *du/ud* pairs. Not all *du/ud* pairs are feasible. For example, if a definition occurs to generate some condition under which a use will never occur, then this *du* pair can never occur. An MTE fault which can never occur is called a *redundant* fault.

2 . Simulation. The hardware description is simulated with test vectors. All definition and use occurrences are recorded during the simulation in the form of a timed trace.

3 . MTE Fault Coverage Computation. The timed trace is analyzed to identify all *du/ud* pairs which are executed within *delta* time units of each other. If a *du/ud* pair associated with an MTE fault is executed within *delta* time units of each other, then the MTE fault is considered to be detected. The ratio between the number of detected MTE faults and the total number of MTE faults is the MTE fault coverage.

V. Experimental Results

To evaluate the MTE fault model, we have used SystemC as the hardware-software language, although any language which supports discrete event simulation might have been used. The design examples used are taken from the SystemC web site [23]. Table I provides general information on the examples used. Test vectors are provided with the benchmarks. Table I includes information on the number of lines of code, the total number of *du/ud* pairs, the number of *du/ud* pairs executed during simulation, and the MTE fault coverage. The fault coverage numbers shown assume that $\delta = \text{inf}$. The number of *du/ud* pairs executed counts only those pairs which are executed without an intervening definition (definition-clear path). In the example "stmach", most of the *du/ud* pairs are redundant, so the true maximum coverage should be higher than 0.46. The reason for redundant in "stmach" example is that the definitions and uses are enclosed within conditional branches which are mutually exclusive. For example, signal "key" is defined when defining signal "state" to $state_a$, and some use of "key" occurs when "state" is not in $state_a$, so this *du* pair is a redundant *du* pair because it will never have an opportunity to occur.

benchmark	# of lines	# of du/ud pairs	du/ud pairs executed	fault cov. without threshold
fir	187	18	16	0.89
bus	78	16	16	1
simplex	192	24	17	0.71
stmach	195	186	86	0.46

TABLE I
GENERAL INFORMATION OF BENCHMARKS.

The detailed timing analysis of each example is listed in separate tables because the *clock* signal used for each example is different, and therefore causes large differences in timing relationships. To eliminate the effect of redundant *du/ud* pairs on our evaluation, when computing fault coverage in Tables II-V, we consider only the *du/ud* pairs which are executed assuming that $\delta = \inf$. The coverage values presented in Tables II-V are normalized using the maximum coverage values in Table I.

In our experiments, we required that each *du/ud* pair is executed twice or more because each process executes once on initialization and many *du/ud* pairs only execute at that time. The column labeled "X" is the number of *du/ud* pairs which execute only once. The first row of tables II-V are the minimum time distance of *du/ud* pairs; the second row is the number of *du/ud* pairs executed within the corresponding time distance. Row 3 is the MTE fault coverage corresponding to different time threshold.

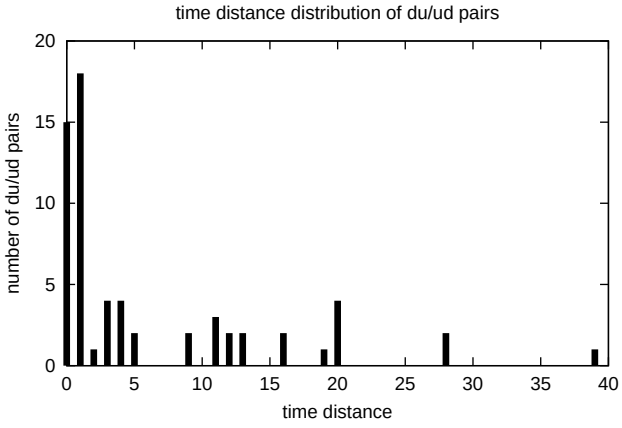

Fig. 6. Time distance distribution of stmach example.

We notice that most of the executed *du/ud* pairs are within a small range of time distance, referred as the *center region*. Other pairs are scattered in a larger range of time distance, referred as the *scatter region*. In the "stmach" example, the number of *du/ud* pairs executed within time distance 0 and 1 is more than a half of all executed *du/ud* pairs. Figure 6 shows the time distance distribution of example "stmach".

VI. CONCLUSIONS

We define a Mis-Timed Event (MTE) fault model which enables efficient evaluation of test patterns for detecting timing-induced functional errors. We provide MTE fault coverage results for several SystemC examples to demonstrate the utility of the approach in identifying potential timing faults in hardware systems. However, more investigation is needed to identify infeasible *du/ud* pairs, and to identify the minimum time distance of each *du/ud* pair.

REFERENCES

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland. Logic verification via test generation. *IEEE Transactions on Computer-Aided Design*, 7(1):138–148, January 1988.

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

[3] S. Chakraborty and D. L. Dill. Approximate algorithms for time separation of events. In *International Conference on Computer-Aided Design*, November 1997.

[4] 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*, SE-15(11):1318–1332, 1989.

[5] A. Dasdan, D. Ramanathan, and R. K. Gupta. Rate derivation and its applications to reactive, real-time embedded systems. In *Design Automation Conference*, pages 263–268, 1998.

[6] A. Dasdan, D. Ramanathan, and R. K. Gupta. A timing-driven design and validation methodology for embedded real-time systems. *ACM Transactions on Design Automation and Electronic Systems*, 3(4), 1998.

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

time distance	X	0	1	4	5	6	total
# of du/ud pairs	1	6	6	1	1	1	16
norm. MTE fault cov.	-	0.33	0.67	0.72	0.78	0.83	-

TABLE II
MTE FAULT COVERAGE ANALYSIS OF EXAMPLE "FIR".

time distance	X	0	1	5	28	total
# of du/ud pairs	-	6	7	2	1	16
norm. MTE fault cov.	-	0.33	0.72	0.83	0.89	-

TABLE III
MTE FAULT COVERAGE ANALYSIS OF EXAMPLE "BUS".

time distance	X	0	5	15	20	total
# of du/ud pairs	1	7	6	1	2	17
norm. MTE fault cov.	-	0.29	0.54	0.58	0.67	-

TABLE IV
MTE FAULT COVERAGE ANALYSIS OF EXAMPLE "SIMPLEX".

t. dis.	X	0	1	2	3	4	5	9	11	12	13	16	19	20	28	39	tot.
pairs #	23	15	18	1	4	4	2	2	3	2	2	2	1	4	2	1	86
ft. cov.	-	.08	.18	.18	.20	.23	.24	.25	.26	.27	.28	.30	.30	.32	.33	.34	-

TABLE V
MTE FAULT COVERAGE ANALYSIS OF EXAMPLE "STMACH".

[8] 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*, pages 666–671, June 1999.

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

[10] P. G. Frankl and J. E. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[11] 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*, pages 158–163, June 1998.

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

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

[14] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[15] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and J.-Y. J. Lu. Fault-simulation based design error diagnosis for sequential circuits. In *Design Automation Conference*, June 1998.

[16] S. Kang and S. A. Szygenda. Design validation: Comparing theoretical and empirical results of design error modeling. *IEEE Design & Test of Computers*, 11(1):18–26, Spring 1994.

[17] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Engineering*, 21(7):685–718, 1991.

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

[19] K. L. McMillan and D. L. Dill. Algorithms for interface timing verification. In *International Conference on Computer Design*, 1992.

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

[21] 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*, 5(2):99–118, April 1996.

[22] S Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, SE-11(4):367–375, April 1985.

[23] SystemC Web Site. *http://www.systemc.org/*.

[24] 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*, 3(2):201–214, 1995.

[25] W. Wolf. *Computers as Components Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2001.

[26] T.-Y. Yen, A. Ishii, A. Casavant, and W. Wolf. Efficient algorithms for interface timing verification. In *European Design Automation Conference*, 1994.

[27] 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.

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