ATPG for Timing-Induced Functional Errors on Trigger Events 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 consider timing-induced functional errors in inter process communication. We present an Automatic Test Pattern Generation (ATPG) algorithm for the co-validation of hardware-software systems. Events on trigger signals (signals contained in the sensitivity list of a process) implement the basic synchronization mechanism in most hardware-software description languages. Timing faults on trigger signals can have a serious impact on system behavior. We target timing faults on trigger signals by enhancing a timing fault model proposed in previous work. The ATPG algorithm which we present targets the new timing fault model and provides significant performance benefits over manual test generation which is typically used for co-validation.
Keywords: Hardware-software, Co-validation, Automatic Test Pattern Generation, Timing validation

1. Introduction

Hardware-software systems are pervasive in the electronics systems industry. 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. In order to manage the complexity of the problem, co-validation techniques in which functionality is verified by simulating (or emulating) a system description with a given test input sequence are being considered. Verification techniques which verify functionality by using formal techniques like model checking, equivalence checking etc. have been explored. Formal verification techniques can guarantee 100% fault coverage but are highly complex and often intractable, whereas co-validation techniques can only provide a degree of certainty which is less than 100%. The complexity of co-validation can be made tractable by using a test sequence of reasonable length, and the degree of certainty can become arbitrarily close to 100%.

Hardware-software systems often have inter-process timing constraints which must be satisfied to ensure correct operation. Co-validation techniques used with hardware description languages like VHDL and Verilog are still developing. Software testing techniques used with behavioral software languages have been well known. One might think that previous work done on software testing may be used to address the co-validation problem. Behavioral hardware description languages including VHDL and Verilog support time-varying *signals*, and include concurrency constructs such as the *process* statement in VHDL.

Previous work [5] considered the timing faults in inter process communication [5] [4]. The Co-design Finite State Machine (CFSM) model [1] has been used to capture the system behavior, and to express the interactions between system components. Signals in a CFSM can be either a *trigger* signals or a *value* signals. Value signals cannot cause the transition, but can be used to choose among different possibilities involving the same set of trigger events. Events on trigger signals form the basic synchronization mechanism in CFSM because only an event on a trigger signal can cause a state transition to occur. Timing faults on value signals are considered in previous work, but timing faults on trigger signals have not been addressed.

The co-validation process typically requires a time-consuming manual test generation step. An Automatic Test Pattern Generation (ATPG) tool which can be used to greatly reduce the time required for co-validation has been developed. The result of the ATPG process is a timed sequence of events on the system inputs which will detect timing-induced faults based on an extension to the fault model described in [5]. 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: Section 2 describes proposed design fault model for timing-induced errors of trigger signals. Section 3 outlines the stages of a test pattern generation technique to target the proposed timing fault model for trigger signals and the timing fault model for value signals presented in [4]. Results are presented in Section 4 and the work is sum-

marized in Section 5.

2. Timing Fault Model

Timing faults exists if the signals are assigned values at incorrect time even though the values are correct. There are several signal timing relationships which must be maintained to guarantee correct communication between the two processes. Dataflow fault models have been extended to capture timing-induced functional defects [5]. A timing fault is associated with the *definition* and *use* of a signal in the behavioral description. A definition of a signal x is an assignment of a value to x, and a use of x is the assignment of another signal y which depends on the value of x. For example, $a <= in1$ is a definition of a and $z <= a$ is a use of a. A timing error can occur if a definition-use pair are executed in the incorrect order. For example, if signal a should be assigned to a constant before it is used, but due to a timing problem, a is used before it is properly assigned. We refer to this type of fault as a **Mis-Timed Event Late** (MTE_{late}) fault because the definition occurs later than it should. Conversely, an MTE_{early} fault occurs on a definition-use pair if the definition is executed too early.

The MTE fault model was originally proposed in [5] and it has been adapted for the CFSM model here. Intuitively, a *trigger signal* is one which is contained in the sensitivity list of a process. An event on a trigger signal causes the execution of a process in time. Trigger signals implement the basic synchronization mechanism in HDLs including VHDL and Verilog.

A **Mis-Timed Event of a Trigger signal (MTET)** fault to be associated with definition and use statements on a given signal $s \in S$ along with the state either preceding or succeeding the use, where S is the set of all signals used in the design. The existence of an MTET fault indicates that the associated signal definition occurs at the incorrect time and causes the system to be in a different state. Two types of MTET faults can exist, $MTET_{early}$ where the definition occurs earlier than the correct time, and $MTET_{late}$ where the definition occurs later than the correct time.

An early fault on a trigger signal can cause the event to occur while the receiving CFSM is in a state prior to the corect state. This can be explained with the help of an example shown in Figure 1. This example has two processes X and Y which run in parallel and exchange data through a FIFO buffer. The *read* and *write* signals are trigger signals because events on these signals cause the state of the FIFO to change. In order for the system to operate correctly, an event on the *read* signal cannot occur when the FIFO is in the **empty** state, and an event on the *write* signal cannot occur when the FIFO is in the **full** state. An MTET fault can cause events on the *read* and *write* signals to occur in the **empty** and **full** states respectively.

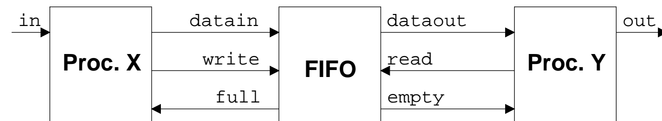

Figure 1: Two processes communicating via a FIFO

2.1. Detection of Timing Faults

The 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. We define an *error span threshold, δ, a non-negative integer representing the maximum time between the definition and use occurrence.*

3. Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) deals with generating a timed sequence of input vectors which causes the detection conditions of the timing fault under consideration to be satisfied. Test generation flow for the ATPG tool is shown in the Figure 2. The process is started with hardware-software description of the design under consideration. A fault list is generated. For each feasible path set, one undetected fault is chosen at a time. A test pattern to detect the selected fault is generated if one exists. Fault simulation is performed to determine whether the generated test pattern detects any other fault in addition to the fault under consideration. This process of selecting an undetected fault and trying to generate a test vector is repeated for every path set. The following sections describe the various stages of the ATPG tool in detail.

Berkeley's Software Hardware Interface Format (SHIFT) which can represent various Co-design Finite State Machines (CFSMs) is used to specify the system design to the ATPG tool. SHIFT is one of the intermediate formats in Berkeley's POLIS tool [2]. It can be obtained directly from the ESTEREL to SHIFT compiler which is readily available in POLIS. ESTEREL is a synchronous programming language, which is devoted to programming control-dominated software or hardware reactive systems.

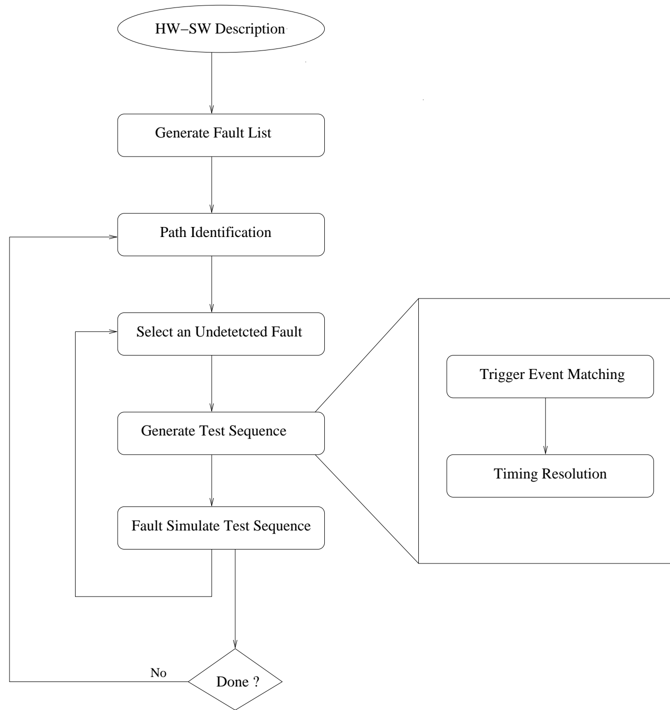

Figure 2: Test Generation Flow

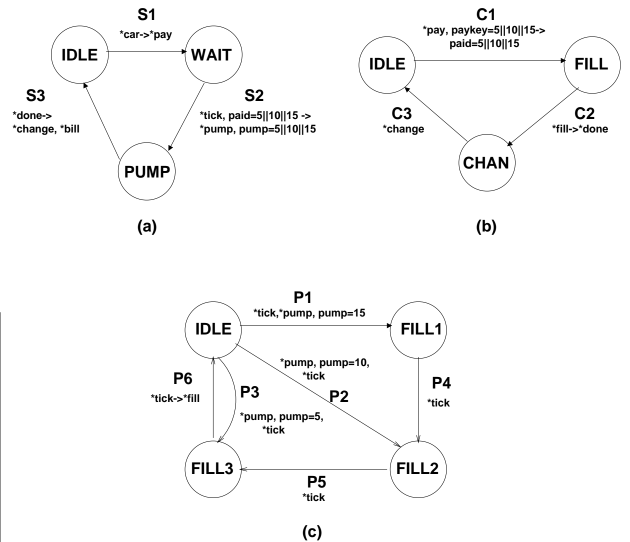

Figure 3: The Gas Station Problem, (a) Station CFSM, (b) Customer CFSM, (c) Pump CFSM

Signal	Early	Late
$paid$	3	3
$*pay$	3	3
$*pump$	9	9
$pump$	3	3
$*fill$	3	1
$*change$	1	3
$*done$	3	1

Table 1: Faults in Gas Station Example

The example in Figure 3 is based on the gas station problem [6]. The gas station problem is a simulation of an automated self-serve gas station. Our version of the gas station consists of three tasks station, customer and 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 the station, customer and the pump tasks are shown in Figure 3. Fault list for the trigger signals is generated based on the fault model explained in Section 2. Fault list for the value signals is generated based on the fault model presented in [4]. The number of faults associated with each signal is shown in the Table 1.

The set of paths selected must satisfy the detection requirements of a fault, and the paths must be compatible [4]. Two paths are incompatible in a computation if they cannot both exist in the same computation due to conflicting edge triggering requirements ,i.e, if the number of uses of a signal s is greater than the number of definitions of that signal s in the path set, the paths are incompatible. Our algorithm for path identification is enumerative. All sets of paths whose length is less than a fixed limit are explored successively. For example if the maximum length is 4, every path in the path set has a length less than or equal to 4. Path sets which are infeasible are not explored further.

3.1. Test Sequence Generation

Test pattern generation identifies a sequence of events on input signals which will cause a given definition-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 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

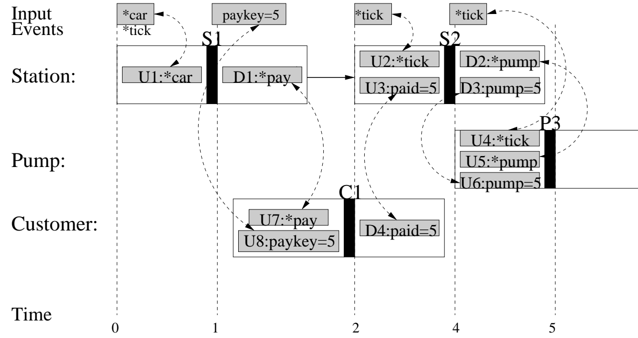

Figure 4: Trigger event matching and Timing resolution

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 input signals.

Trigger Event Matching - *Events which cause an edge to be traversed in a CFSM must be matched with the corresponding definition (trigger) event. This matching is required to ensure that each CFSM traverses the specified paths [4].*

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 [4]. All signal definitions and triggers which are matched during trigger event matching must be mapped to the same time step. Restrictions must be placed on the timing to ensure that unspecified edges are not traversed.*

Figure 4 shows a path set with uses of the signals matched with the corresponding definitions. The matched signals are shown with dotted arrows. The events are placed in time for a delay of 1 and a clock period of 2. The timing problem is expressed as a linear program. For example, consider the edge C1 in Figure 4. The timing problem for the edge C1 can be expressed as $C1 > S1 + delay$, $C1 + delay \leq S2$.

3.2. Fault Simulation

Once the test sequence for a timing-induced fault under consideration is detected, this fault is marked detected and all the other faults which are not detected are checked whether they can be detected by this test sequence or not. If detected they are marked detected. A byproduct of the test sequence generation process is a complete timed computation which contains the time of each event.

Example	# of CFSMs	# of faults	Fault Coverage
Gas Station	3	48	72.92%
Seat Belt Controller	2	94	62.77%
Traffic Light Controller	3	30	63.33%
Railroad Crossing	4	22	81.82%
Lift Controller	4	313	50.48%

Table 2: Benchmark Examples

4. Results

A version of the Automatic Test Pattern Generation (ATPG) tool has been developed. The ATPG tool is tested to observe the variation in complexity and fault coverage with different parameters. The variation of CPU time per fault with delta (error span) is analyzed. Fault coverage is not 100% due to the inclusion of redundant faults in the fault list. The effect of redundant faults is discussed.

Figure 2 shows the stages involved in the ATPG tool. Five examples have been used to test the ATPG tool. The linear programming formulation in timing resolution stage was solved using the mathematical tool lp_solve. The ATPG tool has been used to detect each of the MTE faults. Each ATPG run varied on at least one of four parameters: ML - *Maximum length of the path,* CLK - *the period of the clock,* δ (delta) *- 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 2 shows the characteristics of the benchmark examples and fault coverage for* ML=*6,* CLK=*2,* delta=*10,* delay=*1. The results are presented in the form of plots showing the variation in CPU time, fault coverage with varying parameters. Section 4.1 discusses redundant faults. Section 4.2 summarizes the results.*

Figures 5, and 6 show how the CPU time and fault coverage vary with varying parameters for the gas station example. It is shown that fault coverage becomes constant as the max path length becomes large. Max path length limits the ATPG search of the solution space by restricting the search to short computation sequences. Max path length is needed to limit CPU time, just as CPU time and backtracking limits are used to restrict ATPG for stuck-at faults. As max path length is increased, the ATPG search becomes more complete and more faults are detected. The fault coverage becomes constant when all non-redundant faults are de-

tected. Figures 7 and 8 show how the CPU time and fault coverage vary with varying parameters for the traffic light controller example.

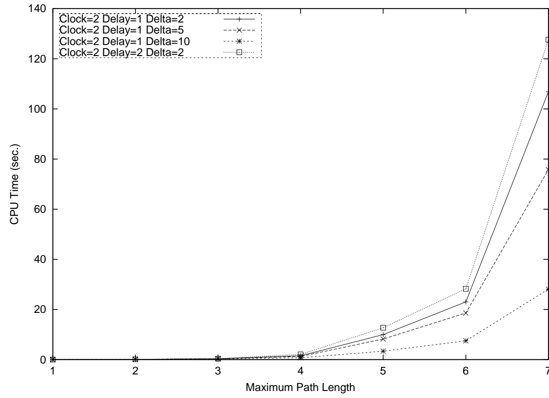

Figure 5: **Gas Station: CPU Time Vs Maximum Path Length**

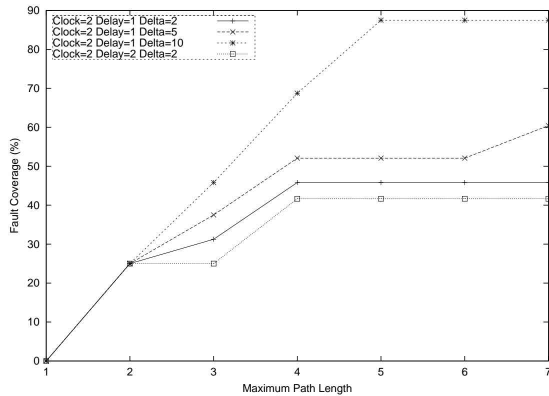

Figure 6: **Gas Station: Fault Coverage Vs Maximum Path Length**

4.1. Redundant Faults

Fault coverage as can be seen from the above tables is not 100%, this is because of the way the fault list is generated. All the combinations of definitions and uses of the signal are considered. But some definition-use combinations can never occur. This can be shown with an example in figure 9. Definition D2 and use U5 of the signal ∗pump can never occur together as definition D2 occurs with definition D3 (pump $= 5$*) and use U5 occurs with use U6 (pump* $= 10$*) and as can be seen use U6 and definition D3 are incompatible.*

4.2. Summary

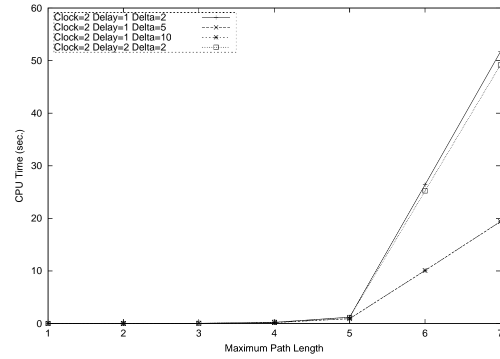

Figure 7: **Traffic Light: CPU Time Vs Maximum Path Length**

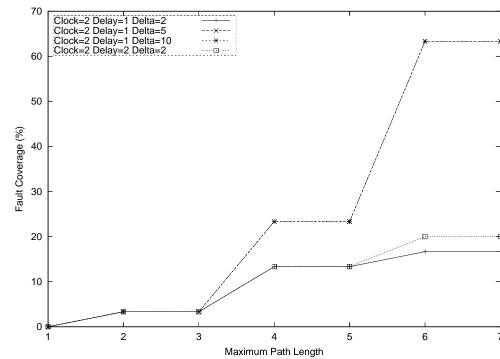

Figure 8: **Traffic Light: Fault Coverage Vs Maximum Path Length**

The ATPG tool developed is tested with five examples. Different results have been obtained by varying the parameters clock period, delay, delta and maximum path length. The results are summarized as follows,

- *Fault coverage increases with maximum path length and becomes constant after a particular maximum path length. This is also the case with increasing delta when other parameters are kept constant. The increase in fault coverage is due to the increase in number of path sets traversed which in turn increases the probability of the faults being detected. After a particular max path length the fault coverage becomes constant as the faults left are redundant faults and are anyway undetectable.*

- *CPU time increases with maximum path length. This is due to the exponential increase in number of feasible path sets traversed.*

- *CPU time decreases significantly in most of the cases as delta increases with other parameters kept constant. The decrease in CPU time is due to the decrease in time taken by the linear program solver*

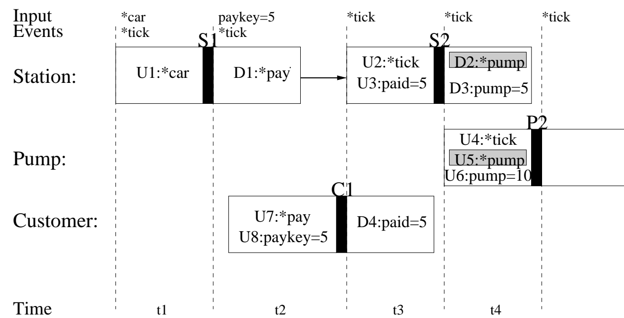

Figure 9: **Example for a Redundant Fault**

as the constraints become more relaxed increasing delta.

- *CPU time changes slightly in some cases as the time taken by path identification and time taken by trigger event matching dominate the time taken by the linear program solver.*
- *Fault coverage decreases as the value of clock period and delay become closer.*
- *CPU time per faults detected in most of the cases decreases with increasing delta.*

5. Conclusions

We present an automatic test pattern generation technique for the co-validation of hardware-software systems. A new fault model specifically targeted towards the CFSM model is used. Timing faults involving trigger signals which are ignored previously are considered. An unique ATPG tool which targets the timing-induced functional errors at behavioral level is developed. One of the key features of this ATPG tool is its interface to Berkeley's POLIS [2] SHIFT format. An heuristic in the form of maximum path length i s given as an input to the system to make the complexity tractable in the average case. Efficient techniques have been used to prune out infeasible path sets restricting the search space.

6. 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] S. Arekapudi, F. Xin, J. Peng, and I. Harris, "Test Pattern Generation for Timing-Induced Functional Errors in Hardware-Software Systems," *IEEE International High Level Design Validation and Test Workshop*, 2001.

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

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