ATPG for Timing Errors in Globally Asynchronous Locally Synchronous Systems*

Srikanth Arekapudi, Fei Xin, Jinzheng Peng, and Ian G. Harris

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003, USA

Globally Asynchronous, Locally Synchronous (GALS) systems are now commonplace in many cost-critical and life-critical applications, thus motivating the need for a systematic approach to verify functionality. The complexity of the verification problem for large heterogeneous GALS systems necessitates the development of simulation-based validation approaches to uniformly validate hardware, software, and their interaction. GALS systems are comprised of several processes which may be mapped to different hardware and software components and communicate through asynchronous interfaces. Communication between these processes must be verified to ensure that the system is working correctly. Previous work focusses on checking the correctness of individual processes rather than communication between multiple processes. Timing errors may cause a signal to have an incorrect value for a short time period. Timing errors can cause a problem in GALS systems if the value of a signal with a timing error is used while is has an incorrect value. This paper presents an automatic test pattern generation (ATPG) tool to generate tests for timing-induced functional errors.

1. Introduction

The high density of VLSI circuits has increased the difficulty of the clock routing problem to the point that even a single chip must be composed of multiple clock domains, which are either semi-synchronous or asynchronous. Systems of this nature are referred to as Globally Asynchronous, Locally Synchronous (GALS) systems. The majority of complex systems, even single chip systems, are implemented as GALS systems. The widespread use of GALS systems in cost-critical and life-critical applications motivates the need for a systematic approach to verify functionality. Several obstacles to the verification of GALS systems make this a challenging problem. Formal techniques such as model checking require properties to be specified manually. The level of sophistication required to develop a complete and correct set of properties in a reasonable amount of time is beyond the abilities of many designers. In order to manage the complexity of the problem, validation techniques in which functionality is verified by simulating (or emulating) a system description with a given test input sequence are being considered. The tractability of simulation-based validation makes it the only feasible verification solution for large designs.

GALS systems are sensitive to timing problems at the asynchronous interface between components. A timing error at an asynchronous interface can result in incorrect functionality. Behavioral hardware description languages, including VHDL and Verilog, support time-varying *signals*, and include concurrency constructs such as the *process* statement in VHDL. Timing validation at component interfaces is central to the GALS validation problem.

*This work was supported in part by the National Science Foundation under award CCR-0204130

The 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 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 [1]. The test generation tool uses the Codesign Finite State Machine (CFSM) model to describe system behavior. 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 simulation-based validation is presented in Section . Section introduces the computational model used to describe the behavior of GALS systems. Section describes the design fault model used for timing errors. Section outlines the stages of the test pattern generation technique. Results are presented in Section and results are discussed in . Conclusions and future work are presented in Section .

2. Previous Work

2.1. *Validation Fault Models*

Several fault models have been developed to evaluate behavioral designs, many of which are based on software test fault models. Mutation analysis describes a fault model which was originally developed in the field of software test [4,5], but has also been applied to hardware validation [6] and to manufacturing test [7] (referred to as micro-operation faults). 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 number of validation fault models are based on the traversal of paths through the CDFG representing the system behavior. The earliest control-dataflow fault models include statement coverage, branch coverage and path coverage [8] models used in software testing. Statement coverage associates a potential fault with each line of code, and requires that each statement in the description be executed during testing. It is well accepted 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. The branch coverage metric associates potential faults with each direction of each conditional in the CDFG. 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.

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 [9,10,11,12] to alleviate this weakness. The OCCOM fault model has been applied for hardware validation [9,10] and for software validation [11]. The OCCOM approach inserts faults called *tags* at each variable assignment which represent a positive or negative offset from the correct signal value.

2.2. *Validation Test Generation*

Several automatic test pattern generation approaches have been developed which vary in the search space technique adopted, the fault model assumed, and the design abstraction level used.

A challenge in generating test vectors for validation of behavioral/RTL designs is that those designs are typically composed of complex *arithmetic units*, such as adders/subtractors, multipliers, barrel shifters, etc., as well as *Boolean functions.* To alleviate the problem, Fallah *et al.* [10] proposed a *hybrid* satisfiability approach, HSAT, which considers linear arithmetic constraints together with boolean SAT constraints. Nonlinear functions, such as multiplication, used in a behavioral description are linearized and modeled as integer linear equations.

Constraint propagation techniques between different domains have been also explored in [13] to generate test vectors and check assertions on HDL descriptions using publicly available logic program solvers. The common denominator of all these methods is that an explicit backtracking or constraint propagation between the heterogeneous domains (arithmetic and Boolean) is employed. Another approach to the problem resolves both linear and Boolean constraints in the single unified domain by formulating a problem as an Integer Linear Program (ILP) [14], or as a Constraint Logic Programming [15]. These methods offer a significant improvement in terms of the complexity of the designs they can handle and the runtime.

Several techniques have been also developed for test sequence generation using randomized algorithms to improve overall coverage without a strongly directed search mechanism. An example of such a technique is presented in [16,17] which uses a genetic algorithm to successively improve the population of test sequences. Work presented in [18] uses a Random Mutation Hill Climber (RMHC) algorithm which randomly modifies a test sequence to improve a testability cost function.

We have previously applied both domain testing and dataflow testing methods to the validation of behavioral VHDL descriptions [19,20]. We have also proposed a design fault model for timing errors [1], and we have presented preliminary results using the model.

3. Codesign Finite State Machines

Codesign Finite State Machines (CFSMs) [2] are the formal model used in the POLIS codesign tool for specification, simulation, analysis, synthesis and optimization. The CFSM model is used to describe the behavior of GALS systems by our ATPG tool. A system is described as a network of CFSMs, each of which describes a single concurrent process. Each edge in a CFSM is a cause-reaction pair, where the cause describes the events which cause the edge to be traversed, and the reaction describes the events emitted when the edge is traversed. There is a non-zero amount of time between the cause events which trigger a state transition and the emission of the reaction events. Communication between processes involves writing to and reading from queues which are located at the inputs of the receiver.

3.1. *Semantics of CFSMs*

A system is described as a network of CFSMs. Each CFSM is a finite state machine with extensions to add support for data handling and asynchronous communication. A CFSM network has,

- a finite state machine part that contains a set of inputs, outputs, states, a transition relation and an output relation,
- a data computation part in the form of references in the transition relation to datapath computation.
- executes a transition by producing single output reaction based on a single snapshot input assignment in 0 time. Once a reaction starts, it must continue to the end before the next reaction to some other input assignment can start. This is locally synchronous behavior.
- reads inputs, executes a transition and produces outputs in an unbounded but finite amount of time as seen by the rest of the system. This is asynchronous communication from system perspective.

This semantics, along with a scheduling mechanism to coordinate CFSMs, provides a Globally (at system level) Asynchronous and Locally(at CFSM level) Synchronous (GALS) communication model. Each process in the network is a CFSM.

3.1.1. *Classes of Signals*

CFSMs communicate through signals. Signals can be any of the following three types,

- trigger event with value
- trigger event
- pure value

Events on trigger signals can be detected and used only at the time when they are emitted, and each event can only be consumed once by each CFSM. Trigger signals implement the basic synchronization mechanism of CFSMs because events on trigger signals can cause transitions in other CFSMs. Pure value signals cannot directly cause a transition, but can be used to choose among different possibilities involving the the same set of trigger events.

3.2. *Gas Station CFSM Example*

To clarify the the CFSM model we present a CFSM network representing the Gas Station Problem [21] in Figure 1. The gas station problem is an implementation of an automated self-serve gas station. Our version of the gas station consists of three tasks,

- Station
- Customer
- Pump

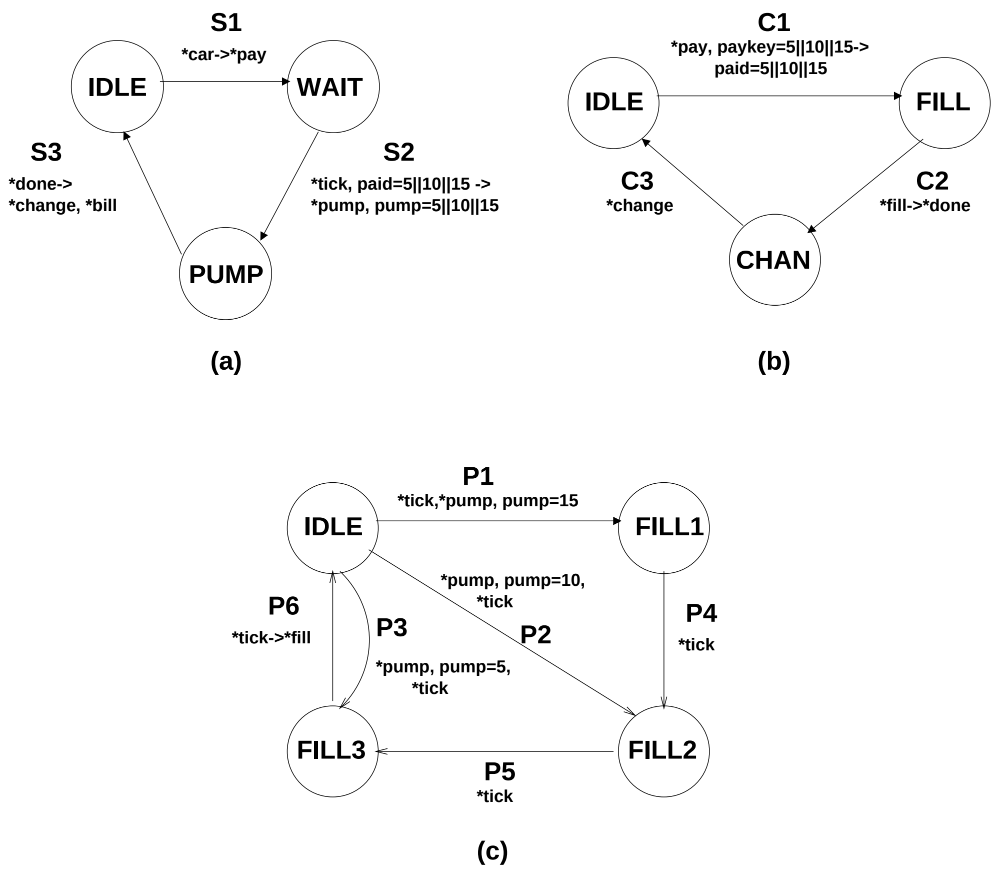

Figure 1: The Gas Station Problem, (a) Station CFSM, (b) Customer CFSM, (c) Pump CFSM

The pump provides a discrete amount 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 1. Each edge in the CFSMs is labeled. The $*tick$ signal is the output of a clock. The signals in the gas station problem shown in Figure 1 are classified into three types,

- trigger event with value - $*pump, pump$
- trigger event - $*car, *tick, *done, *pay, *bill, *change, *fill$
- pure value - $paid, paykey$

4. Timing Design Faults

A *design defect* is a difference between the design and the intended specification. Examples of design defects include,

- incomplete or inconsistent specifications
- incorrect mappings between different levels of design
- violations of design rules

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 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.

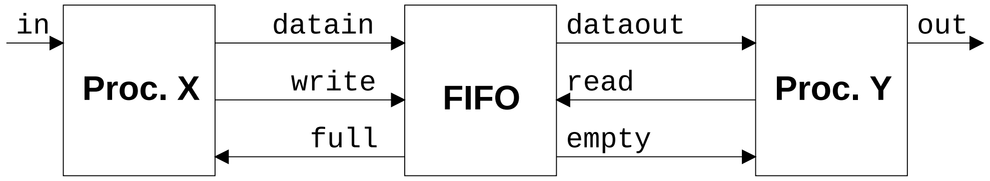

Figure 2: Two processes communicating via a FIFO

A timing fault exists if a signal is assigned a value at the incorrect time. A timing fault will cause a signal value to endure for the incorrect length of time, and therefore can be observed only during the incorrect time period. The detection properties of timing faults are explained with the help of a small example shown in Figure 2. This example has two processes X and Y which run in parallel and exchange data through a FIFO buffer. 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.

4.1. *Timing Fault Model*

Dataflow fault models have been extended to capture timing-induced functional defects in previous work [1]. Dataflow fault models classify each signal occurrence in a behavioral description as either a definition occurrence or a use occurrence. A definition occurrence of a signal 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. For example the VHDL statement $x \Leftarrow y$ represents the definition of x and use of y. Each definition and use occurs at a discrete point in time and, due to a design fault, may occur either early or late.

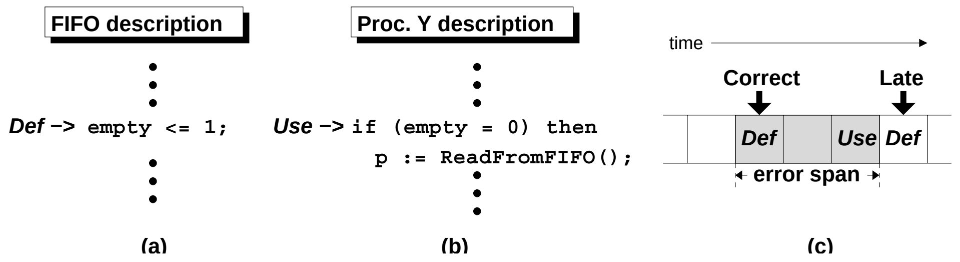

Figure 3: *empty* signal is asserted late, (a) a section of the FIFO description, (b) a section of the process Y description, (c) event trace with error span highlighted.

Figure 3 depicts the timing details involved with a late *empty* signal. Figure 3a shows the definition of the *empty* signal in the FIFO description where *empty* signal is asserted to 1. Before process Y can read data from the FIFO, it must check the *empty* signal as shown in Figure 3b. The event trace shown in Figure 3c 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.

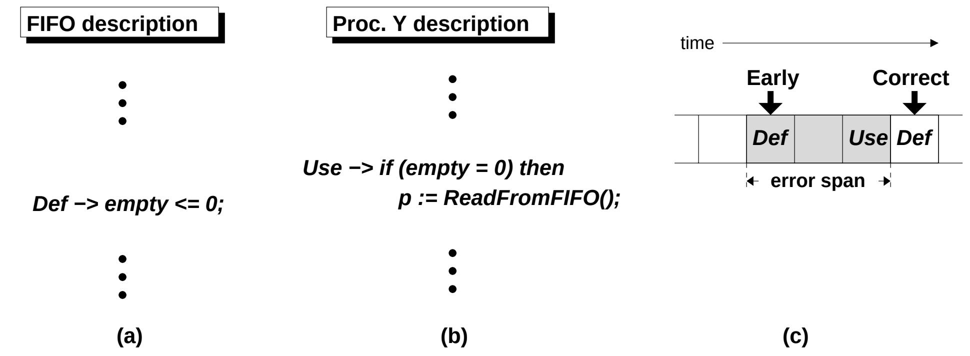

Figure 4: *empty* signal is asserted late, (a) a section of the FIFO description, (b) a section of the process Y description, (c) event trace with error span highlighted.

Figure 4 depicts the timing details involved with an early *empty* signal. Figure 4a shows the definition of the *empty* signal in the FIFO description where *empty* signal is asserted to 0. Before process Y can read data from the FIFO, it must check the *empty* signal as shown in Figure 4b. If *empty* signal is asserted to 0 earlier than it should be, process Y tries to read data from the empty FIFO. The event trace shown in Figure 4c shows both the correct and the early assertion times of the *empty* signal.

Figures 3 and 4 show examples of *Mis-Timed Event of a Value signal (MTEV)* faults. A potential MTEV fault is associated with each pair of definition and use statements on a given signal $s \in S_v$, where S_v is the set of all value signals used in the design. The existence of an MTEV fault indicates that the associated signal definition occurs at the incorrect time and causes the associated use to receive incorrect data. Two types of MTEV faults can exist, $MTEV_{early}$ where the definition occurs earlier than the correct time, and $MTEV_{late}$ where the definition occurs later than the correct time. Figure 3 and 4 respectively show $MTEV_{early}$ and $MTEV_{late}$ faults associated with the *empty* signal.

4.1.1. *Fault Model for Trigger Signals*

Because events on trigger signals implement the synchronization mechanism in CFSMs, timing faults on trigger signals have a different effect than timing faults on value signals. A *Mis-Timed Event of a Trigger signal (MTET)* fault is associated with definition and use statements on a given signal $s \in S_t$ along with the state either preceding or succeeding the use, where S_t is the set of all trigger 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 correct state. This can be explained by examining the FIFO example shown in Figure 2. 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.

4.2. *Detection of Timing Faults*

The example of Figure 3 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.

4.3. *Error Span Threshold*

The error span is the different between the correct definition time and the erroneous definition time. In order to ensure the detection of MTE faults, the time difference between the associated definition and use must be less than the error span. The error span cannot be known for a particular fault apriori so some threshold must be assumed for the purpose of test generation. We use δ to refer to the maximum time allowed between a definition-use pair associated with a fault which is detected. The δ value is used to constrain the test generation process for each fault to ensure that the definition-use pairs associated with each fault are separated by no more than δ time units. This ensures that all detectable faults with error span greater than or equal to δ are detected. Faults with an error span less than δ may not be detected because the definition and use are not guaranteed to occur within the error span.

The value of δ has several impacts on the test generation process. The δ value constrains test generation so a small δ tends to increase test generation time since constraints are more tight. The δ value impacts fault coverage because some faults may not be detectable for small values of δ if the minimum time separation between the definition-use pair is greater than δ. This implies that fault coverage should increase as δ increases because more definition-use pairs may be executed with δ time units. The impact of δ on both test generation time and fault coverage are demonstrated in the results in Section .

The value of δ also impacts the sensitivity of the validation process to small timing variations. In order for an MTE fault on a value signal to be detected, the execution order of the definition and use must be reversed due to a change in timing. A small change in the timing of the definition may not be sufficient to reverse the order of the definition and use.

5. 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. For each fault, the ATPG tool identifies a computation of the system whose output response is different in the presence of the fault. During a computation a system must traverse a single path in each CFSM in the system. Any computation can be described as a *path set*, a set of paths which describe the state transitions in each CFSM. The ATPG tool explores the space of all path sets to identify those which detect faults in the system. A path set alone is not sufficient to describe a computation because it lacks timing information. Each transition in each path must be placed in time in such a way that the semantics of the CFSM model are satisfied.

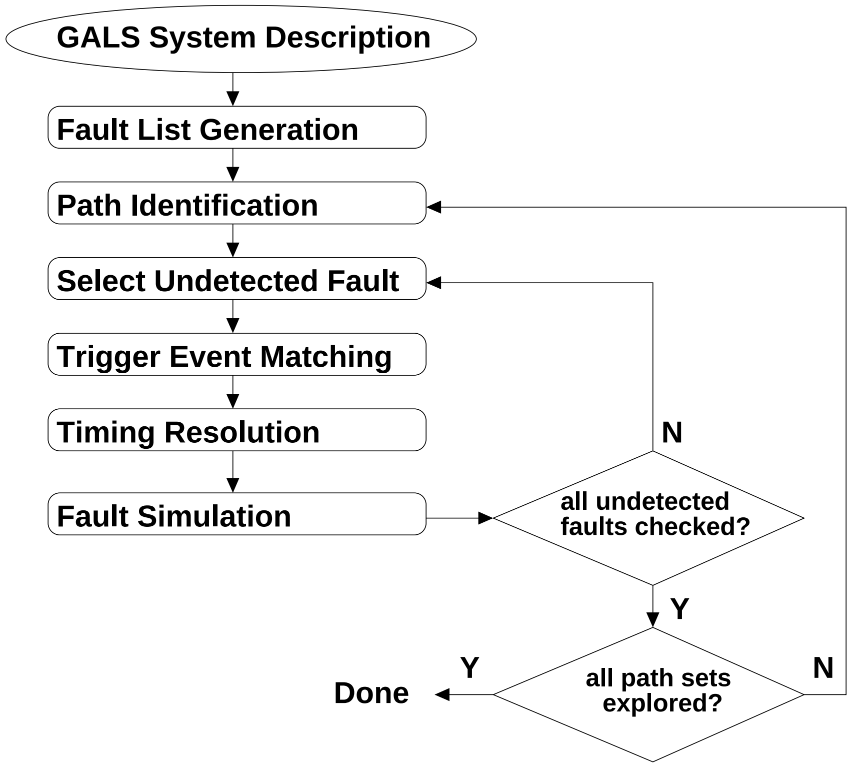

Figure 5: Test Generation Flow

The test generation flow of the ATPG tool is shown in Figure 5. The input to the process is the description of the design under test in the form of a network of CFSMs. A list of faults is generated based on the definitions and uses of each signal in the description. The algorithm contains two main loops, the outer loop which iterates through all feasible path sets, and the inner loop which determines which faults can be detected by each path set. The *Path Identification* step generates the next feasible path set. The *Trigger Event Matching* and *Timing Resolution* steps map each transition in the path set to a point in time, completing the test sequence for the chosen fault. Fault simulation is performed with the test sequence to determine which faults in addition to the targeted fault are detected. The inner loop continues until all untested faults have been evaluated and the outer loop continues until all feasible path sets within a fixed path length limit have been evaluated.

5.1. *GALS System Description*

Berkeley's Software Hardware Interface Format (SHIFT) which can represent various Codesign 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 ESTEREL [3] code using the ESTEREL to SHIFT compiler which is readily available in POLIS. ESTEREL is a synchronous programming language, which is devoted to programming control-dominated reactive systems.

5.2. *Fault List Generation*

All timing faults are enumerated and placed in a fault list which will be used during test generation to select faults to target. Timing faults are associated signal definitions and uses, and CFSM edges. Signal definitions in a CFSM are associated with the reaction of some edge, and signal uses are associated with the cause of some edge. We enumerate the set of all timing faults in a CFSM network by applying the definitions presented in Section . As an example we enumerate some of the timing faults in the gas station problem shown in Figure 1.

- **Early Faults in Value Signals** ($MTEV_{early}$)

  The *paid* signal has 3 possible values, 5, 10, and 15. The signal *paid* has 3 definitions (one for each value) in the customer CFSM, and has 3 uses in the station CFSM. There are a total of 3 definition-use pairs of the *paid* signal which involve the same signal value, so there are 3 early faults associated with the *paid* signal.

- **Late Faults in Value Signals** ($MTEV_{late}$)

  Each definition-use pair of the *paid* signal is associated with a single late fault, so there are 3 late faults associated with the *paid* signal.

- **Early Faults in Trigger Signals** ($MTET_{early}$)

The $*fill$ signal is a trigger signal used by one edge of the customer CFSM, and defined by one edge in the pump CFSM. There is only one definition-use pair involving $*fill$. The source state of the edge using $*fill$ has 3 incoming edges. This means that there are 3 $MTET_{early}$ faults associated with the $*fill$ signal, one for each definition-use-incoming edge triple.

- **Late Faults in Trigger Signals** ($MTET_{late}$)

  There is only one definition-use pair involving $*fill$. The destination state of the edge using $*fill$ has 1 outgoing edge. This means that there is 1 $MTET_{early}$ fault associated with the $*fill$ signal which corresponds to the single definition-use-outgoing edge triple.

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

All the faults associated with the gas station example are listed in Table 1. There are a total of 48 faults.

5.3. *Path Identification*

All possible path sets containing paths whose length is under a fixed limit are generated and evaluated for compatibility. If all paths in a path set are compatible with each other then the timing of the path set is further refined during the *Trigger Event Matching* and *Timing Resolution* stages of test generation.

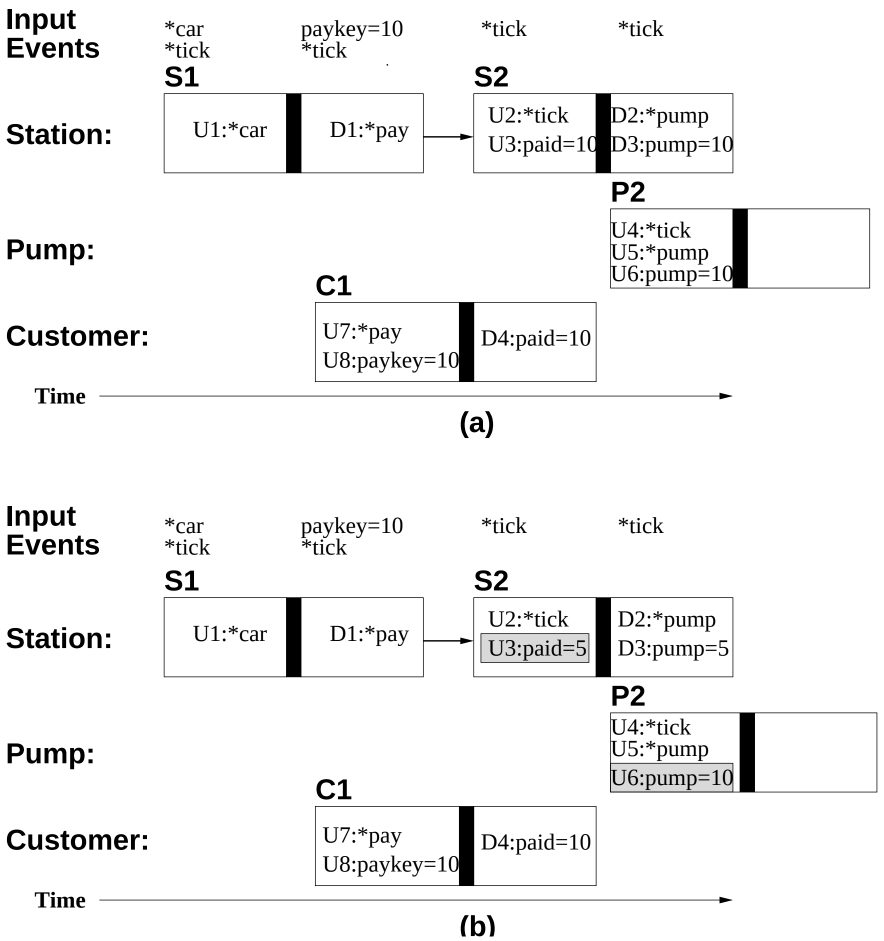

Figure 6: Example path sets, (a) Compatible Paths, (b) Incompatible Paths

Signal	Uses	Definitions
*car	1	Primary Input
paykey=10	1	Primary Input
*tick	2	Primary Input
*pay	1	1
*pump	1	1
pump=10	1	1
paid=10	1	1

Table 2: Uses and Definitions for Compatible Paths

Two paths are incompatible if they cannot both exist in the same computation due to conflicting edge triggering requirements. We identify two types of conflicts. The first type involves the number of definitions and uses of a trigger signal. The number of uses of a trigger signal in a single CFSM must be less than or equal to the number of definitions of that signal. This restriction arises from the fact that a trigger event is instantaneous and may be consumed only once by each CFSM. Figure 6 shows two path sets for the gas station CFSM network shown in Figure 1; one path set is compatible and the other is incompatible. The top row contains the input events which include the definitions of the $*tick$ signal representing a clock, the $*car$ signal, and the *paykey* signal. The successive three rows depict the paths traversed in each of the three CFSMs in the system. Each path is composed of a set of path elements which represent the edges in traversed in the corresponding CFSM. Each path element is depicted as a rectangle with two sections; the left section lists the causes of the corresponding edge and the right section lists the reactions of the edge. Each path element is labeled with the name of its corresponding CFSM edge. The number of definitions and uses of each signal in the path set in Figure 6a is shown in Table 2. The table shows that the path set is compatible because the number of definitions and uses satisfy the feasibility constraints. The path set in 6b is incompatible because the $paid = 5$ and $pump = 10$ events are used, as highlighted, but are never defined.

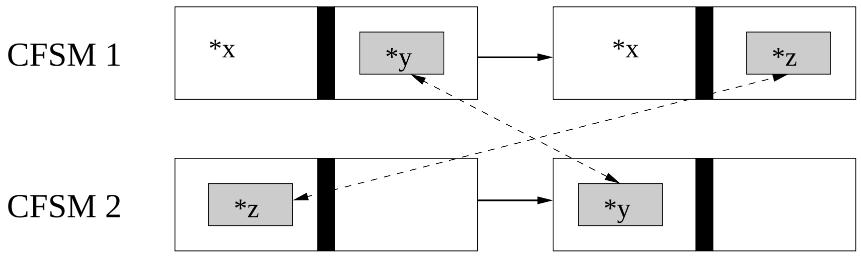

Figure 7: Path sets having cross definition-use pairs

Two paths are also incompatible if the order of definitions of a set of trigger signals in one path does not match the order of uses of those signals in the other path. An example of a path set which violates this requirement is shown in Figure 7. In this example CFSM 1 defines $*z$ after defining $*y$ while CFSM 2 uses the signals in the reverse order making the path infeasible.

5.4. *Trigger Event Matching*

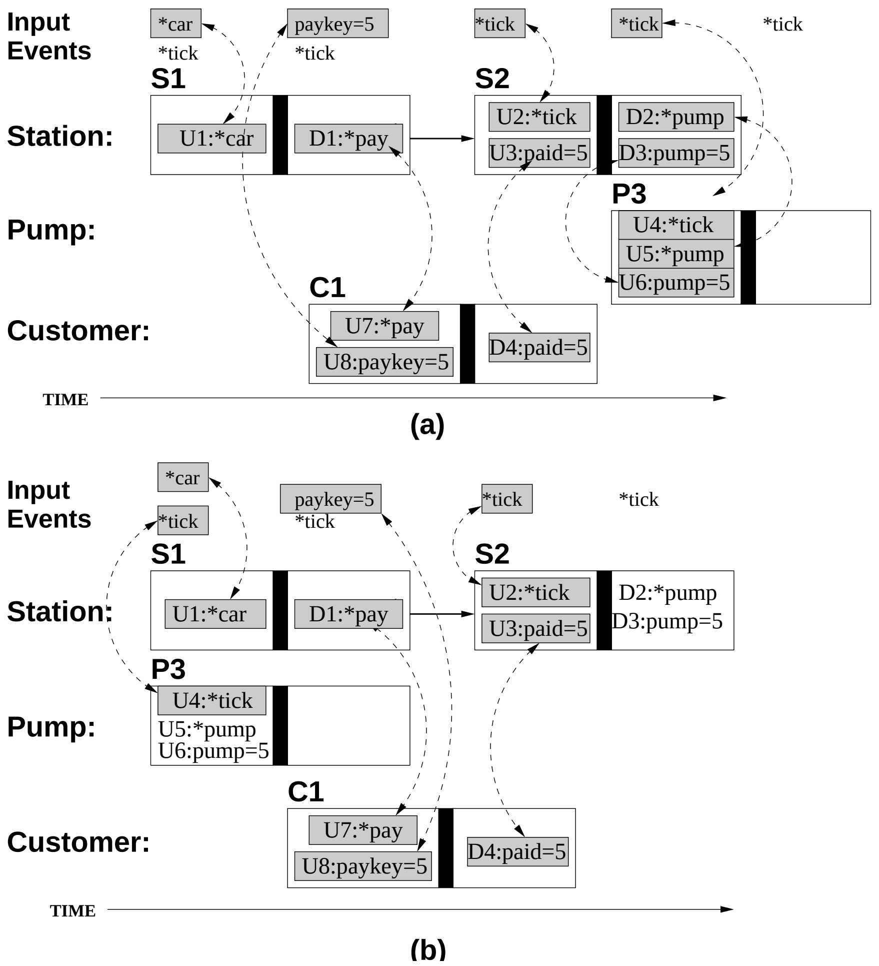

Figure 8: Trigger Matching, (a) Feasible Trigger Matching , (b) Infeasible Trigger Matching

Events which cause an edge to be traversed in a CFSM must be matched with a corresponding definition (trigger) event. 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 8(a). The top row contains the input events which include the definitions of the $*tick$ signal representing a clock. The successive three rows depict the paths traversed in each of the three CFSMs in the system. Each path element is labeled with the name of the CFSM edge to which is corresponds in Figure 1. Three paths are shown in the Station CFSM {S1, S2}, the Pump CFSM {P3}, and the Customer CFSM {C1}. Each matched definition-use pair is indicated by a dashed line with arrows at each end indicating the associated definition and use. Figure 8(a) shows matching which is feasible because all uses are matched to a corresponding definition. Figure 8(b) shows matching which is infeasible because the uses of $pump = 5$ and $*pump$, which are the causes of path element P3, are cannot be matched to definitions.

5.5. *Timing Resolution*

Each event which triggers a CFSM edge must be mapped to a time step. The final 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. Restrictions must be placed on the timing to ensure that unspecified edges are not traversed. For example, Figure 9 shows a timed path set for the gas station example in Figure 1. The path set in Figure 9 contains an event on the $*tick$ signal in each time step. The Station CFSM traverses two edges, S1 and S2, and during the time between these transitions, the Station is in its WAIT state. The WAIT state is sensitive to events on the $*tick$ signal because the outgoing edge S2 is triggered by the $*tick$ signal. The timing shown in Figure 9 is inconsistent at time step $t2$ because the Station CFSM is in its WAIT state and an event occurs on the $*tick$ signal, but edge S2 is not triggered at that time.

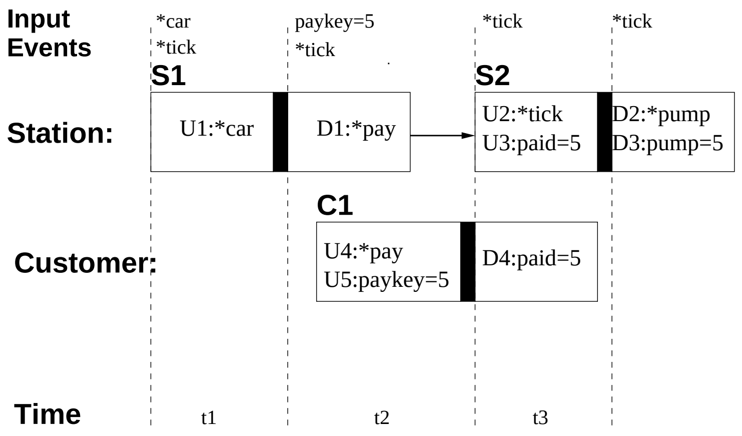

Figure 9: Timing resolution, two events with timing freedom

Timing resolution is formulated as a linear program in which the execution time of each edge is represented as an integer 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 is 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 then a single continuous subrange must be selected for each edge.

5.6. *Fault Simulation*

Fault simulation consists of determining the set of all faults which are detected by a new test sequence so that the faults can be removed from future consideration. Once a test sequence is completed for a timing fault under consideration, the fault is marked detected. Each new test sequence may detect a number of faults other than the one for which it was created. By examining the timed computation, it is simple to locate all faults pairs for which the fault detection criteria presented in Chapter are satisfied.

Example	# of CFSMs	# of faults	Fault Coverage	CPU Time (sec)
Gas Station [21]	3	48	87.50%	7.50
Seat Belt Controller [2]	2	94	62.77%	650.83
Traffic Light Controller [22]	3	30	63.33%	10.10
Railroad Crossing [23]	4	22	81.82%	2.28
Lift Controller [24]	4	313	50.48%	908.59

Table 3: Benchmark Examples

6. Experimental Results

A version of the 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 δ (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.

Five examples have been used to test the ATPG tool. The integer linear programming formulation in timing resolution stage was solved using the public domain tool *lp_solve*. The ATPG tool has been used to detect each of the MTE faults. All results were run using a Sun Ultra 5 machine with 256 Mb of RAM. In our experiments four parameters were varied to evaluate their impacts on ATPG complexity: *ML* - Maximum length of the path, *CLK* - the period of the clock, δ - the error span threshold, and *delay* - the delay associated with each CFSM edge. For simplicity we assume that each edge has the same delay. Table 3 shows the characteristics of the benchmark examples and fault coverage for *ML*=6, *CLK*=2, δ=10, *delay*=1.

6.1. *Detailed Results, Gas Station Example*

This example is shown in Figure 1 and its details are presented Section . Figure 10 shows the detailed result produced to detect the $MTEV_{early}$ and $MTEV_{late}$ faults associated with the definition of $paid = 5$ in edge C1, and the use of $paid = 5$ 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. 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 $MTEV_{early}$ fault is detected when the $*car$ and $paykey = 5$ input events occur some small ϵ time after time steps 0 and 2 respectively because this forces the definition of $paid = 5$ to occur ϵ after the use. Similarly, the $MTEV_{late}$ fault is detected when the $*car$ and $paykey = 5$ input events occur some small ϵ time before time steps 0 and 2 respectively because this forces the definition of $paid = 5$ to occur ϵ before the use.

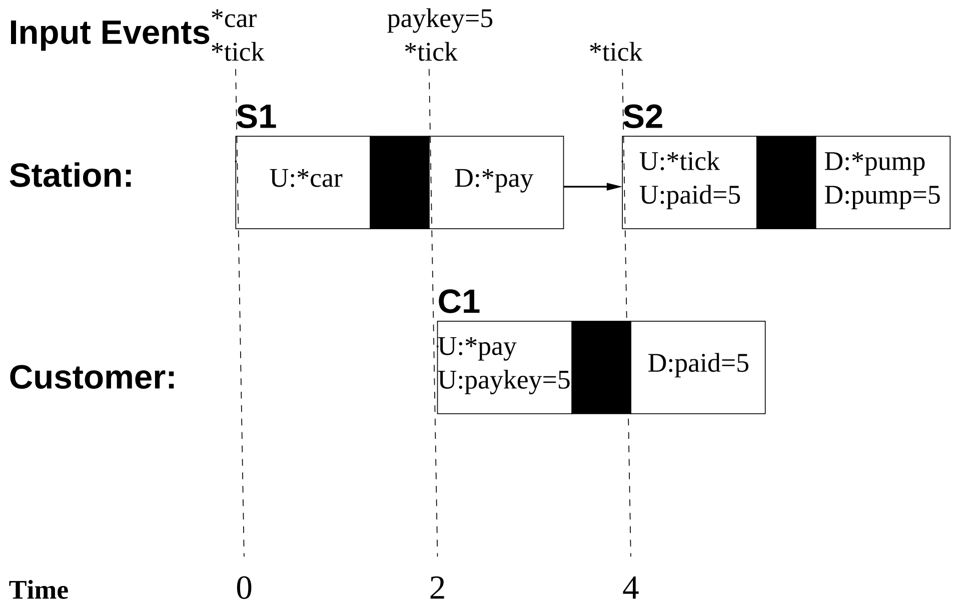

Figure 10: Test Sequence for MTEV faults on $paid = 5$

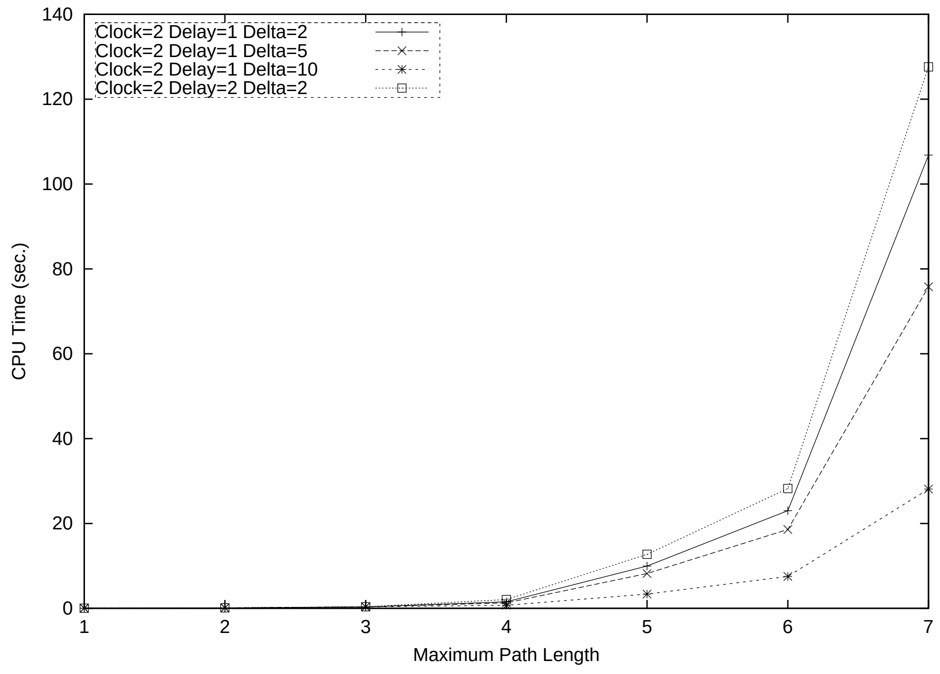

Figure 11: Gas Station: CPU Time Vs Maximum Path Length

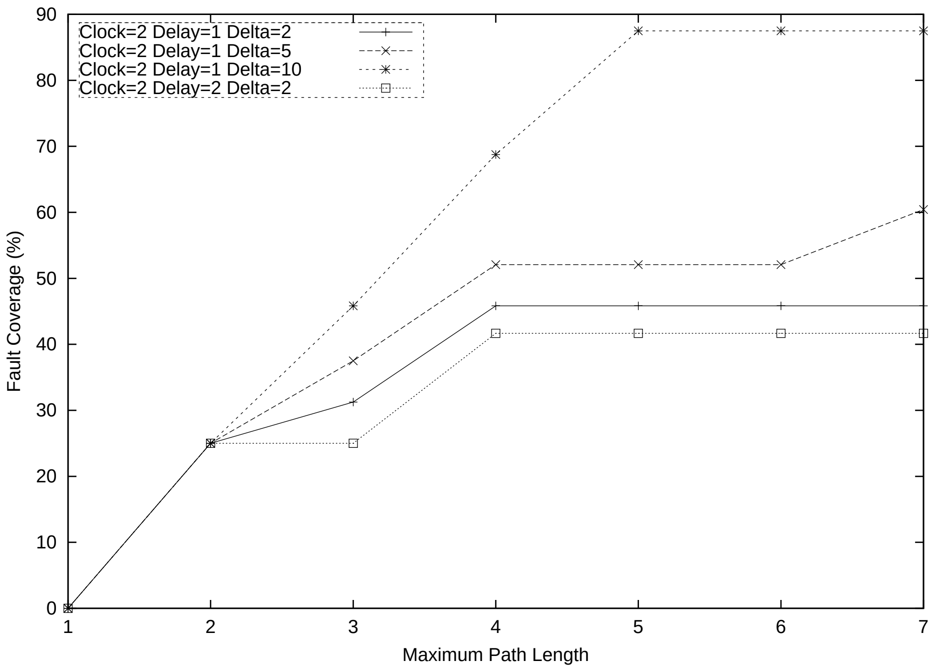

Figure 12: Gas Station: Fault Coverage Vs Maximum Path Length

- Figure 11 shows how the CPU time varies with maximum path length for different values of clock period, delay, and δ. CPU time increases exponentially with maximum path length; this is due to the exponential increase in the number of path sets traversed with maximum path length.
- Figure 12 shows how the fault coverage varies with maximum path length for different values of clock period, delay, and δ. Fault coverage initially increases with maximum path length because some faults are only detectable with longer test sequences. The fault coverage ceases to grow when the maximum path length is large enough to detect all non-redundant faults.

6.2. *Detailed Results, Seat Belt Controller*

An example [2] of seat belt alarm controller is shown in Figure 13. The function of the controller is to beep the alarm for five seconds or until the key is turned off, if the belt has not been fastened within five seconds after the key is turned on. There are two CFSMs; the Controller CFSM is shown in Figure 13(a) and the Timer CFSM is shown in Figure 13(b).

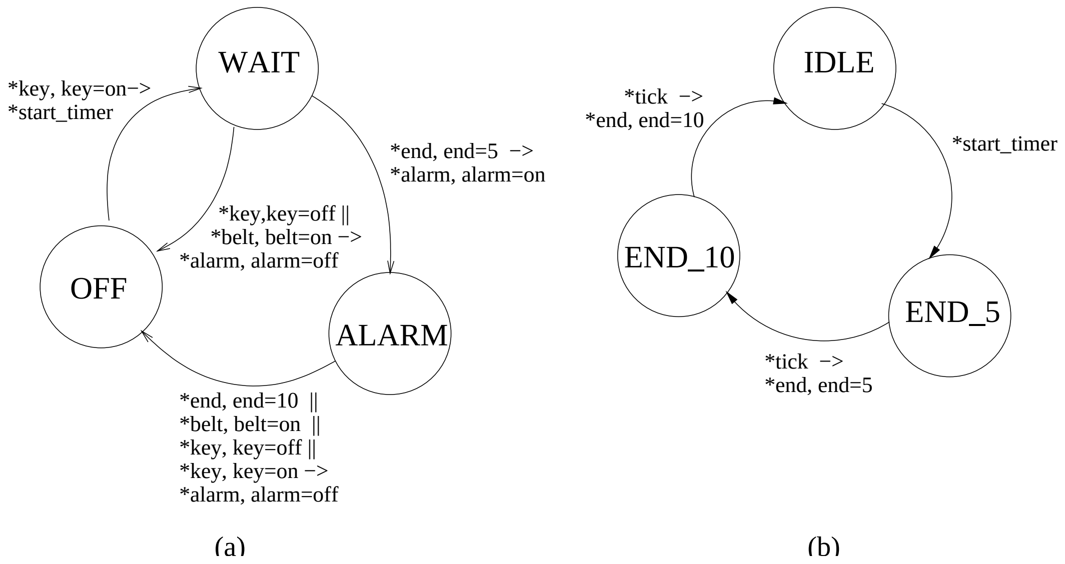

Figure 13: Seat belt controller, (a) Controller CFSM, (b) Timer CFSM

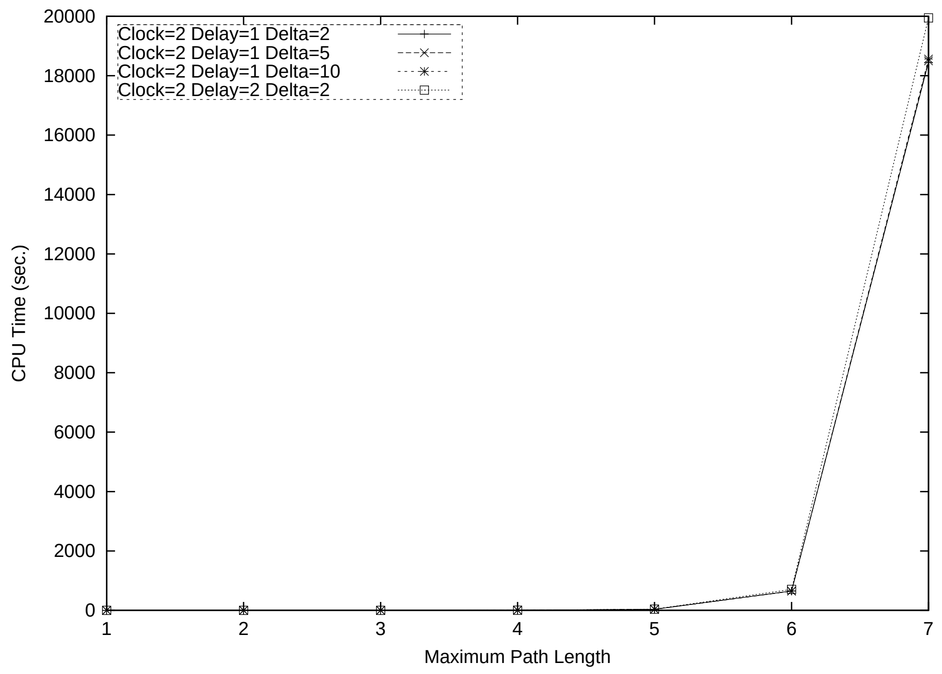

Figure 14: Seat Belt: CPU Time Vs Maximum Path Length

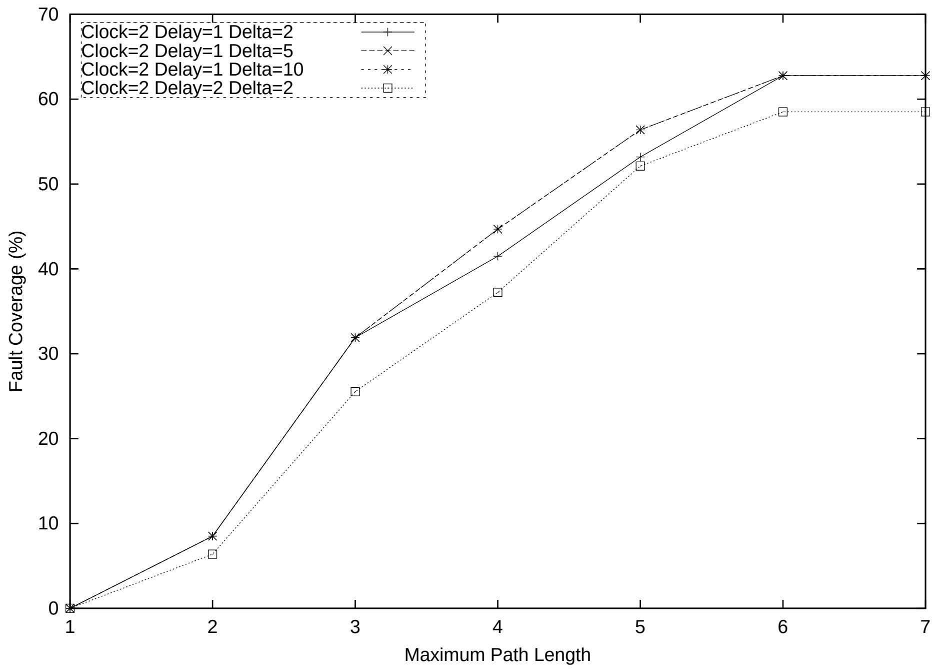

Figure 15: Seat Belt: Fault Coverage Vs Maximum Path Length

- Figure 14 shows how the CPU time varies with maximum path length for different values of clock period, delay, and δ. CPU time increases exponentially with maximum path length.
- Figure 15 shows how the fault coverage varies with maximum path length for different values of clock period, delay, and δ. Fault coverage initially increases with maximum path length and then levels off when all non-redundant faults are detected.

7. Discussion of Results

Several trends can be identified in the results which are common over all benchmark examples.

- Fault coverage increases with maximum path length and becomes constant after a particular maximum path length. Some faults can only be detected by long test sequences, so fault coverage increases until all non-redundant faults are detected.
- CPU time increases with increasing maximum path length, this is due to the exponential increase in number of feasible path sets traversed.
- Fault coverage increases significantly in most of the cases as δ increases with other parameters kept constant.
- 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 as the constraints become more slack with 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.
- CPU time per faults detected in most of the cases decreases with increasing delta.

7.1. *Redundant Faults*

Fault coverage in all cases is less that 100%; this is because all combinations of definitions and uses of the signal are considered as potential faults but some faults are redundant and can never be stimulated. We have identified two classes of redundant faults. We refer to the first type of redundancy as *unmatchable* redundancy because it exists because a definition-use pair on a trigger signal cannot be matched together. Unmatchable redundancy occurs when there exist definition-use pairs which can be matched but which are associated with other definitions and uses which cannot be matched. An example of this can be seen in the path set for the gas station example shown in Figure 16. Definition D2 must occur with definition D3 because they are reactions of the same edge, and use U5 must occur with U6 for the same reason. D3 and U6 cannot occur together because they involve different values of the *pump* signal. D2 and U5 cannot be matched because they are associated with D3 and U6 which cannot be matched. The majority of redundant faults are of the unmatchable type. For example, consider the gas station example with parameters CLK=2, Delay=2, δ=10 and maximum path length of 7. Out of 13 undetected faults, 12 of these faults are unmatchable redundant faults.

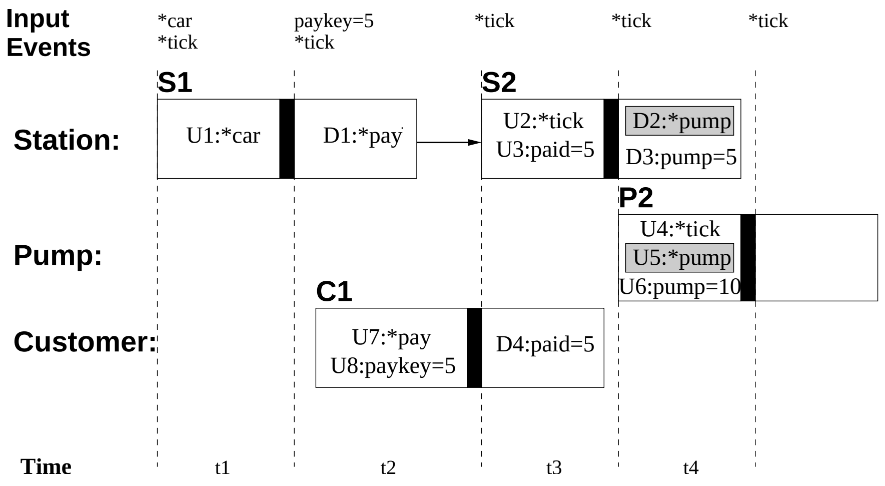

Figure 16: Example of a Redundant Fault

The second type of redundancy is referred to as *error span* redundancy because the size of error span threshold δ is too small to allow the detection criteria of a fault to be satisfied. The detection of an MTE fault depends on the value of δ; the definition and use must occur within δ time steps for a value signal, and the definition-use pair must occur within δ time steps of a state transition for a trigger signal. If the value of δ is too small then it may not be possible to satisfy the timing constraint required for detection. What this means functionally is that the system is not sensitive to a small change in the timing of some definition-use pairs. This type of redundancy is very difficult to identify in practice because it depends on the solution to the Minimum Time Separation problem which is known to be NP-complete. An example of a fault in this class is in the gas station example with parameters CLK=2, Delay=2, δ=10 and maximum path length of 7. Out of 13 redundant faults, one fault is in this class. The fault is an early fault on the definition of $*$*done* in edge C2, the use of $*$*done* in edge S3, and the edge S2 which precedes S3. The definition and use must occur within δ time units of S2, but the minimum time separation is 12 time units. The timed path set which achieves 12 units of time separation is shown in Figure 17.

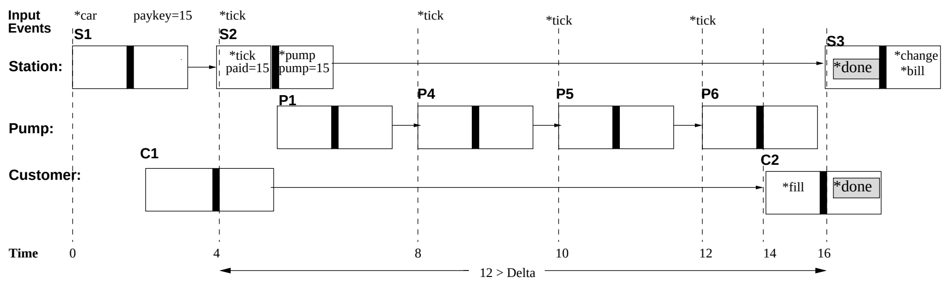

Figure 17: Timed sequence of the path sets having an early fault of $*$*done* Signal

7.2. *Effect of the Error Span Threshold*

The error span threshold, δ, is one of the parameters which is varied during the simulations. We discuss the impact of δ on test generation for the gas station example but similar trends can be seen in other examples as well. Figure 18 shows the variation in the CPU time which decreases with increasing delta. The decrease in time can be explained with the help of Figure 20 which shows the variation in number of feasible matchings considered and Figure 21 which shows the variation in number of calls to the linear program solver. As delta increases the number of feasible matchings considered decreases drastically which in turn decreases the number of calls to the linear program solver. This decrease in calls to the linear program solver decreases CPU time. As the constraints given to the linear program solver are less strict with increasing delta, the time for solving the equations decreases. Figure 19 shows the variation in fault coverage which increases with increasing delta.

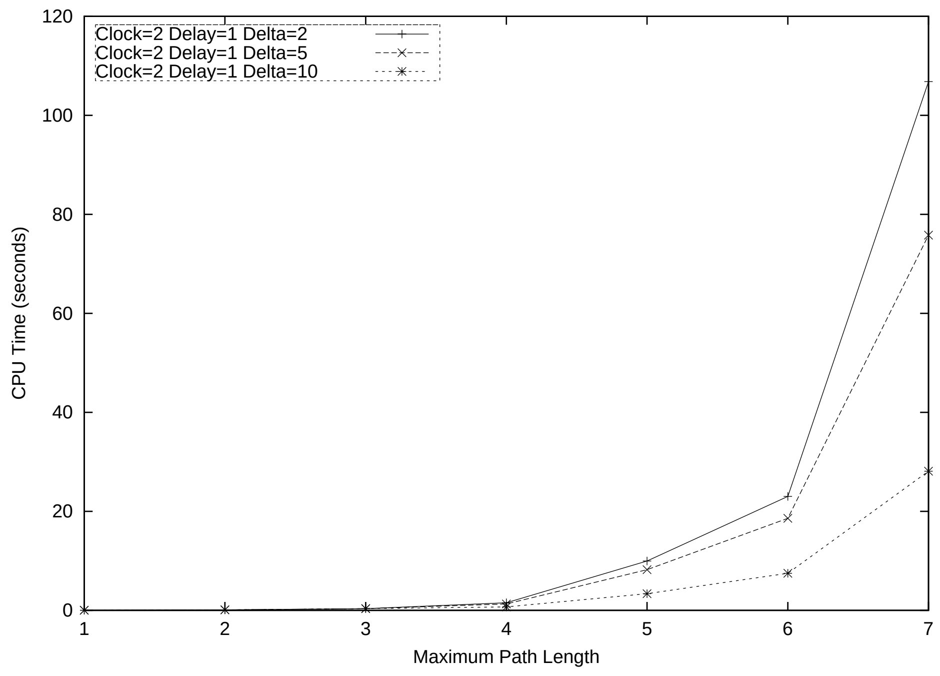

Figure 18: Gas Station: CPU Time Vs Maximum Path Length

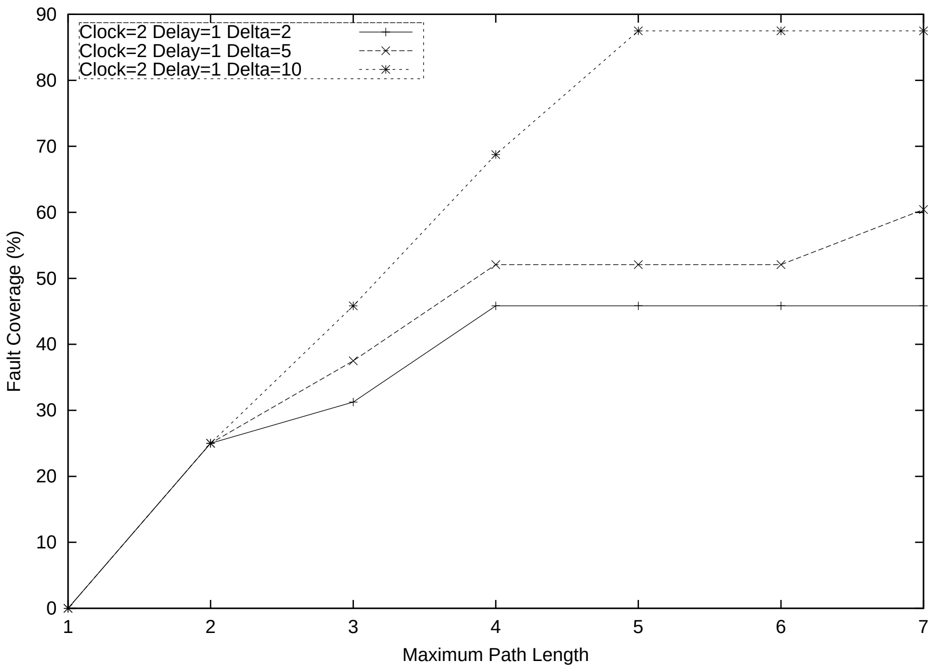

Figure 19: Gas Station: Fault Coverage Vs Maximum Path Length

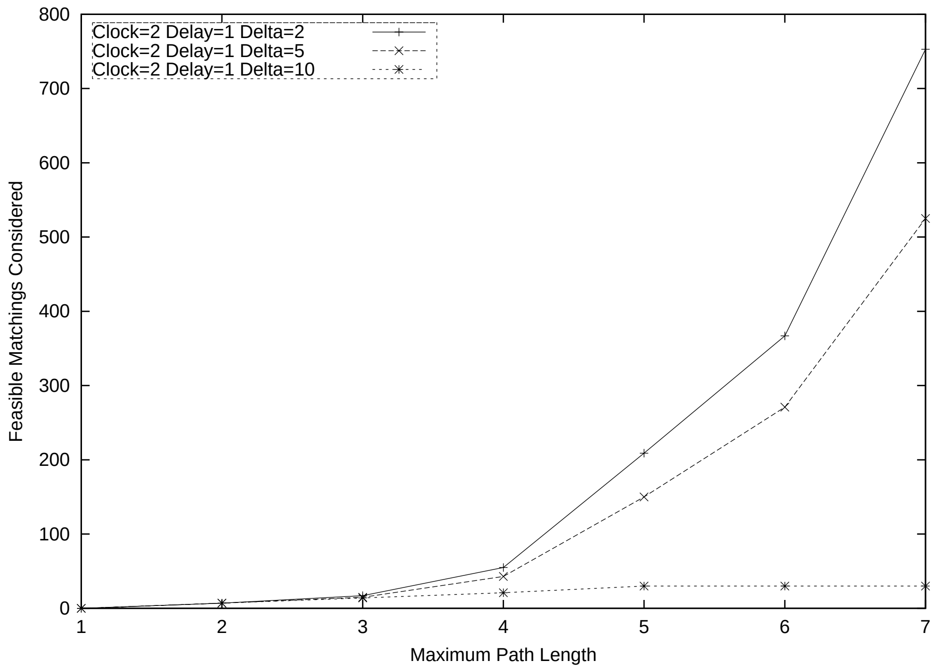

Figure 20: Gas Station: Matchings Considered Vs Maximum Path Length

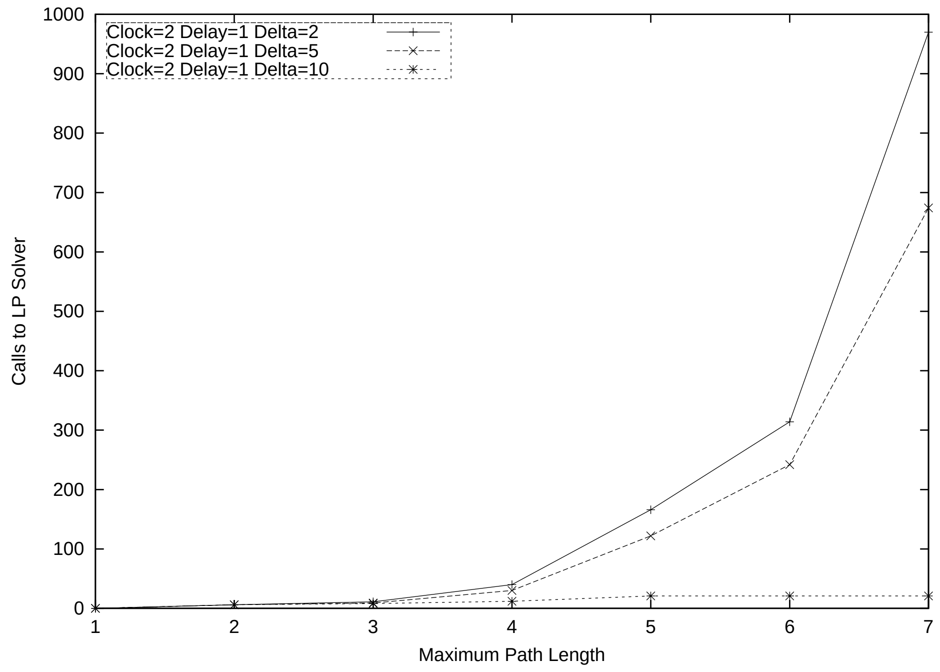

Figure 21: Gas Station: Calls to LP Solver Vs Maximum Path Length

7.3. *Complexity Issues*

The use of an ATPG tool greatly reduces the need for manual interaction in the covalidation process, but the time complexity of the ATPG process must be considered. The experimental results show the actual CPU time required for a number of examples but it is important to understand how time complexity will scale for larger designs. Here we analyze the worst-case time complexity of our ATPG approach as a function of different parameters of the design including its size. We analyze the complexity of the three main steps of test pattern generation, *Path Identification*, *Trigger Event Matching*, and *Timing Resolution*.

7.3.1. *Path Identification Complexity*

The step is enumerative in the worst case, evaluating every possible combination of paths under a given length limit in each CFSM. The number of paths in a CFSM depends on the out degree of each state in the CFSM. An upper bound on number of paths of length i in a CFSM can be expressed as G_c^i, where $c \in C$ is an element of the set of all CFSMs C, and G_c is the maximum out degree of all states in CFSM c. Path identification enumerates all paths whose length is less than or equal to the length limit L, so the total number of paths is the sum of all paths of all lengths less than or equal to L. The upper bound on the number of paths in a CFSM c with length less than or equal to L is expressed as $\sum_{i=1}^{L} G_c^i$.

$$\prod_{c \in C} \sum_{i=1}^{L} G_c^i \tag{1}$$

Path identification explores all path sets which are described by all combinations of paths over the set of all CFSMs. The total number of combinations is the product of the number of paths in each CFSM. The upper limit on the number of path sets is expressed in Equation 1. Based on Equation 1, complexity should increase exponentially with the limit on maximum path length L and this is borne out in the results.

7.3.2. *Trigger Event Matching Complexity*

In the worst case, trigger event matching will attempt to match each signal definition with each use of the same signal with the same value. For a single signal we can express the upper bound on the number of possible matchings as $d_s u_s$, where $s \in S$ is an element of the set of all signals S, d_s is the set of all definitions of signal s, and u_s is the set of all uses of signal s. Trigger event matching must evaluate all combinations of matchings of all signals. The upper bound on the number of matching combinations of all signals is computed as the product of the number of matchings of each signal individually. The upper bound on the total number of matchings is shown in Equation 2.

$$\prod_{s \in S} d_s u_s \qquad (2)$$

The rate at which trigger event matching complexity grows depends on the rate of increase in the number of signal definitions and uses. If we assume that the number of definitions and uses increases as a constant fraction of the number of vertices in the CFSM network, then complexity grows exponentially with the size of the CFSM network.

7.3.3. *Timing Resolution Complexity*

Timing resolution is solved as an integer linear program. The complexity of integer linear programming is known to be $O(k^v)$, where v is the number of boolean variables, and k is a constant determined by the implementation of the solver used. One variable is used to represent each path element in a path set. Since the number of path elements in a path is limited by L and the number of paths is $|C|$, we can express the complexity of timing resolution as shown in Equation 3.

$$O(k^{L*|C|}) \qquad (3)$$

8. Conclusions

A test generation technique is presented to ensure the detection of synchronization faults in globally asynchronous, locally synchronous systems. A novel fault model is used which describes potential timing errors which can cause synchronization faults. The test generation technique constructs test sequences by direct analysis of the underlying codesign finite state machine model used to describe system behavior. The test generation tool developed has been tested with several examples and the results obtained are promising.

1. Q. Zhang and I. G. Harris, "A validation fault model for timing-induced functional errors," in *International Test Conference*, October 2001.
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. 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.
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, no. 2, 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*, October 1996, pp. 885–893.
7. C.-H. Cho and J. R. Armstrong, "B-algorithm: a behavioral test generation algorithm," in *International Test Conference*, 1994.

8. B. Beizer, *Software Testing Techniques, Second Edition*, Van Nostrand Reinhold, 1990.
9. 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.
10. F. Fallah, S. Devadas, and K. Keutzer, “Occom: Efficient computation of observability-based code coverage metrics for functional verification,” in *Design Automation Conference*, June 1998, pp. 152–157.
11. 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*, November 2000, pp. 27–32.
12. 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*, 1999, pp. 182–188.
13. 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, no. 2, pp. 201–214, 1995.
14. Z. Zeng, P. Kalla, and M. Ciesielski, “Lpsat: A unified approach to rtl satisfiability,” in *Design, Automation and Test in Europe Conference*, 2000.
15. Zeng Z., Ciesielski M., and Rouzeyere B., “Functional test generation using constraint logic programming,” in *VLSI-SOC Conference*, 2001.
16. F. Corno, P. Prinetto, and M. Sonza Reorda, “Testability analysis and ATPG on behavioral RT-level VHDL,” in *International Test Conference*, 1997, pp. 753–759.
17. 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*, 2000, pp. 385–389.
18. 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*, 2000, pp. 21–26.
19. Q. Zhang and I. G. Harris, “A domain coverage metric for the validation of behavioral vhdl descriptions,” in *International Test Conference*, October 2000.
20. 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.
21. D. Helmbold and D. Luckham, “Debugging ada tasking programs,” *IEEE Software*, vol. 2, no. 2, pp. 47–57, March 1985.
22. Tiziano Villa, Gitanjali Swamy, and Thomas Shiple, *VIS User’s Manual*.
23. Nikolaj Bjrner, Zohar Manna, Henny Sipma, and Toms E. Uribe, “Deductive verification of real-time systems using step,” *ARTS*, pp. 22–43, 1997.
24. “Call for papers, fourth international workshop on software specification and design,” *ACM SIGSOFT Software Engineering Notes*, , no. 11, pp. 94–96, 1986.