TEST GENERATION FOR HARDWARE-SOFTWARE COVALIDATION USING NON-LINEAR PROGRAMMING

Fei Xin and Ian G. Harris
University of Massachusetts, Amherst MA
fxin@ece.umass.edu, harris@ecs.umass.edu

Abstract

Hardware-software covalidation involves the cosimulation of a system description with a functional test sequence. Functional test generation is heavily dependent on manual interaction, making it a time-consuming and expensive process. We present an automatic test generation technique to detect design errors in hardware-software systems. The design errors targeted are those caused by incorrect synchronization between concurrent tasks/processes whose detection is dependent on event timing. We formulate the test generation problem as a non-linear program on integer variables and we use a public domain finite domain solver to solve the problem. We present the formulation and show the results of test generation for a number of potential design errors.

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. To manage the complexity of the problem, covalidation techniques in which functionality is verified by simulating (or emulating) a system description with a given test input sequence are being considered.

Hardware-software systems are built from separate components which are not globally synchronized. As a result, hardware-software systems are vulnerable to inter-process synchronization problems resulting from timing problems between processes. In previous work we have developed a fault model to describe these timing-induced errors [1] and we have presented a test generation approach for the fault model [2]. Previous research has investigated test generation for hardware-software systems by directly targeting specific fault or by improving fault coverage without targeting individual faults. The problem of targeting the detection of individual faults has been a SAT problem [3] as well as Constraint Logic Programming (CLP) problem [5, 7], solving various engines [4, 6]. Previous work used a Genetic Algorithm [8, 9] and a Random Mutation Hill Climber algorithm [10] to target fault coverage. In this work we present a new test generation approach for the detection of synchronization errors which employs CLP to arrive at a solution. The test generation problem is formulated as a set of non-linear constraints on integer variables. We use a public-domain CLP finite domain solver [6] but this formulation provides the potential to leverage the strength of industrial CLP solvers.

Test Generation Process

The goal of test pattern generation is to identify a timed test sequence of input patterns which will cause the detection conditions of a given timing fault to be satisfied. Figure 1 depicts our test generation process for hardware-software systems. The input of test generation is a system undertest described as a network of Codesign Finite State Machine (CFSMs). The **Computation Constraints Generator** (CCG) is the program which generates a set of computation constraints that describes the behavior of the system under test. To enforce the fault detection conditions, **Fault Detection Constraints** are added to the computation constraints to generate the **Automatic Test Pattern Generation** (ATPG) **constraints**. If a given timing fault can be detected, a test sequence will be identified after solving the ATPG constraints using the public-domain G-Prolog solver [6] .

Synchronization/Timing Fault Model

A synchronization error occurs when a signal has the incorrect value at the time when the signal's value is being used by a process. Synchronization errors can be the result of timing problems at the communication interface between processes. If a signal's value is assigned either earlier or later than expected, it is possible that a process which uses the value will receive an unexpected value. In previous work [1] we have extended traditional data flow fault models to capture timing-induced synchronization errors. 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, $a \Leftarrow in1$ is a definition of a and $z \Leftarrow a$ is a use of a. A timing error can occur if a definition-use pair are executed in the incorrect order. For example, a synchronization error occurs 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

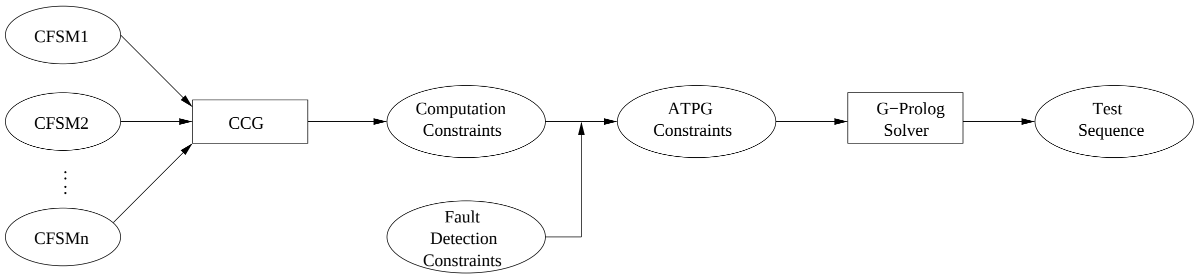

Figure 1: Computation Generator

should. Conversely, an MTE_{early} fault occurs on a definition-use pair if the definition is executed too early.

Behavioral Representation

A behavioral description format must be chosen which represents communication between multiple processes which are not synchronous with each other. The standard finite state machine model is not sufficient for this purpose because it implicitly assumes the existence of a single synchronous component. We have chosen the CFSM model which describes each process as a separate state machine. A system is described as a network of CFSMs, where each CFSM describes a concurrent process in the system. The CFSMs communicate via events on signals. Each event is identified by a name, a value, and a time of occurrence. Each CFSM in a system contains a set of states and a transition relation which can be described as a set of edges in a graph in which each state is represented by a node. Each edge is a *cause-reaction* pair where the cause is a set of event names and values, and each reaction is a set of events and values. When an edge is triggered by an event which matches its cause, the CFSM changes state to the destination state of the edge, and all events in the reaction set are emitted. There is a nonzero time between the cause and the effect which, in practice, would be determined using some performance estimation technique.

As a CFSM example we use the Traffic Light Controller [11] shown in Figure 2. The system contains 3 CFSMs, one representing the highway signal, one representing the road signal, and one representing a timer used to control the lights. Each edge in the CFSMs is labeled $cause \rightarrow reaction$, unless the edge does not involve a reaction in which case only the cause is shown. The highway signal remains green by default. Occasionally, cars from the country road arrive at the traffic signal. The traffic signal for the country road turns green only long enough to let the cars on the country road pass. As soon as there are no cars on the country road, its traffic light turns yellow and then red and the traffic signal on the highway turns green again. A sensor is used to detect cars waiting on the country road. The sensor sets signal *havecar* to be 1 if there are cars on the road; otherwise *havecar*=0. Signal **short* indicates that the time for the highway traffic light to be yellow has ended.

CFSMs include two types of signals, trigger signals and value signals. Trigger signals (denoted with a * prefix) implement the basic synchronization mechanism. Trigger events can be used to cause a transition in a CFSM. This is similar the *sensitivity list* concept in VHDL and other hardware description languages. Value signals may have an arbitrarily large domain and their values persist until the signal value is reassigned. Value signals cannot cause a transition, but can be used to choose among different possibilities. Each edge in a CFSM must be caused by at least one trigger signal.

Synchronization Errors in CFSMs

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

An MTE fault may occur on either a value signal or a trigger signal. An example of an MTE fault on a trigger signal can be seen in example of Figure 2. The $*short$ signal is expected while the HIGHWAY CFSM is in the yellow (Y) state. If there is an MTE_{early} fault on the $*short$ signal causing it to be asserted while the HIGHWAY is in the green (G) state, then the system will deadlock when the HIGHWAY light enters the yellow state.

Detection of Synchronization/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. If the definition and use are close in time,

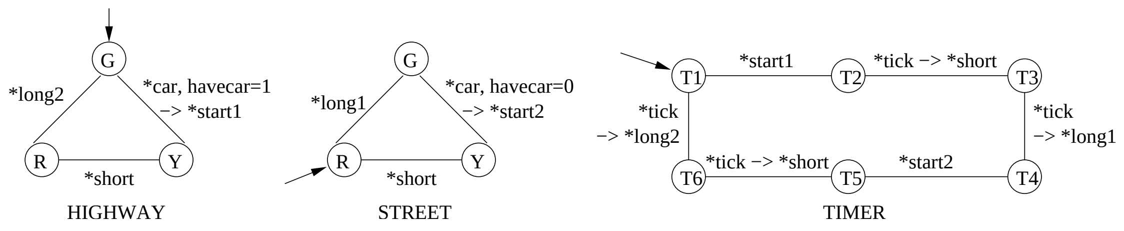

Figure 2: Traffic Light Controller

then a small error in timing will cause the definition and use to be reordered and cause the fault to be detected.

This detection criterion is different for a fault on a trigger signal because the use will move in time with the definition which triggers it. In the system of Figure 2, the assertion of the $*short$ signal can move in time as long as it occurs while the HIGHWAY CFSM is in the yellow state. For this reason, the detection of MTE faults on trigger signals requires that the definition occur when the using CFSM is in an incorrect state. For example, an MTE_{early} fault on the $*short$ signal in Figure 2 is detected if the definition occurs when HIGHWAY is the green state, rather than the yellow state.

Problem Formulation

The test generation problem is described as a set of constraints on the set of variables which represent the computation of the CFSM network. We first describe the set of variables which represent a computation, and we then describe the set of constraints on those equations which ensure fault detection.

Computation Variables

Each feasible computation of a CFSM network is represented by the values of a set of integer variables. The variables used to represent a computation are divided into three categories. Each time step is represented using a distinct set of variables, so each variable describes some aspect of a computation at one time step.

1. **State Variables** - These variables contain the value of the state of a CFSM at a given time step. Each state variable is referred to as $SV_{c,t}$, where c refers to the associated CFSM, and t refers to the time step. The domain of a state variable contains $|S_c|$ values where S_c is the set of all states in CFSM c.

2. **Edge Variables** - The values of these variables refer to the edges in each CFSM which are traversed at a given time step. Each edge variable is referred to as $EV_{c,t}$, where c refers to the associated CFSM, and t refers to the time step. The domain of an edge variable contains $|E_c|+1$ values where E_c is the set of all edges in CFSM c. The domain includes a value which indicates that no edge is traversed at a given time step.

3. **Signal Variables** - These variables collectively contain the values of all signals at a given time step. Each signal variable is referred to as $TV_{p,t}$, where p refers to a signal in the system and t refers to a time step. The domain of a signal variable is the same as the domain of the signal which it represents. Note that the domains of all trigger signals are binary.

Computation Constraints

In this section we define all of the constraints required to ensure that the solution generated correctly satisfies the execution semantics of CFSMs. Constraint equations are all implications of the following form: $antecedent \rightarrow consequent$, where the $antecedent$ is the assignment of a variable to a value and the $consequent$ describes the set of variable assignments which must be asserted to satisfy the semantics of CFSMs. The constraints are divided into three categories based on the type of signal in the antecedent.

1. **State Constraints** - These equations describe the conditions which allow a CFSM to be in a state at a given time step. A CFSM can be in state s at time t if one of the following statements is true.

   (a) The CFSM is in state s at time $t-1$ and the CFSM does not traverse an edge at time $t-1$.

   (b) The CFSM is in a state s_p at time $t-1$ and an edge from state s_p to s is traversed at time $t-1$.

   The equations which express these constraints are produced using the algorithm in Figure 3. In Figure 3 the resulting constraints are referred to as $stateconstr_{c,s,t}$, where c refers to a CFSM, s refers to a state in that CFSM, and t refers to a time step. In the algorithm, $CFSM$ represents the set of all CFSMs, $TMAX$ is the maximum time step, and $InEdge_s$ is the set of all edges which enter state s.

2. **Edge Constraints** - These equations describe the conditions which allow an edge in a CFSM to be traversed at a given time step. A CFSM will traverse an edge e in that CFSM is all of the following statements are true.

   (a) The CFSM is in state s_p at time t, where s_p is the predecessor state of edge e

```
1 for each $c \in CFSM$ {
2   for each $t < TMAX$ {
3     for each $s \in S_c$ {
4       antecedent = $SV_{c,t} = s$
5       consequent = $(SV_{c,t-1} = s) \cap (EV_{c,t-1} = NULL)$
6       for each $e \in InEdge_s$ {
7         $s_p$ = the predecessor state of $e$
8         condition = $(SV_{c,t-1} = s_p) \cap (EV_{c,t-1} = e)$
9         consequent = consequent $\cup$ condition
10      }
11      $stateconstr_{c,s,t}$ = antecedent $\rightarrow$ consequent
12    }
13  }
14 }
```

Figure 3: Algorithm to Generate State Constraints

(b) All of the trigger conditions of edge *e* are satisfied at time *t*

The equations which express these constraints are produced using the algorithm in Figure 4. In Figure 4 the resulting constraints are referred to as $edgeconstr_{c,e,t}$, where *c* refers to a CFSM, *e* refers to an edge in that CFSM, and *t* refers to a time step. When considering the trigger conditions for an edge we refer to a **trigger pair** (p,v), where *p* is a signal and *v* is a value to which signal *p* must be assigned to trigger the edge. We use p_t to refer to the variable which describes the value of signal *p* at time *t*. Each edge *e* is associated with its predecessor state s_p and a set of trigger pairs T_e, all of which must be satisfied to trigger the edge. The process of creating the *edgeconstr* related to edge *e* in CFSM *c* at time *t* is described on lines 3-11 in Figure 4. Lines 7-9 ensure that all the causes related to edge *e* are satisfied. Lines 12-17 describe the condition when no edge in the CFSM *c* is triggered at time *t*.

3. **Signal Constraints** - These equations describe the conditions which allow a signal to have a given value at a given time step. First we describe the trigger signal constraints. A trigger signal in CFSM *c* will have a value of 1 at time *t* (represented by $tsigconstr_{c,t,1}$) only if at least one edge *e* which emits the trigger signal is traversed at time $t-\delta$, where δ is the delay of the edge; A trigger signal having a value of 0 at time *t* (represented by $tsigconstr_{c,t,0}$) implies that none of these edges is traversed at time $t-\delta$, and is formulated as $\bigcap\ (EV_{c,t-\delta} \neq e)$. The equations which express these constraints are produced using the algorithm in Figure 5. In Figure 5 the resulting constraints are referred to as $tsigconstr_{g,t}$, where *g* refers to a trigger signal, and *t* refers to a time step. $TSIG$ refers to the set of all trigger signals, and g_t refers to the variable representing the value of a trigger signal *g* at time *t*. We refer to the set of edges which

```
1 for each $c \in CFSM$ {
2   for each $t < TMAX$ {
3     for each $e \in E_c$ {
4       antecedent$_e$ = $(EV_{c,t} = e)$
5       $s_p$ = the predecessor state of $e$
6       consequent$_e$ = $(SV_{c,t} = s_p)$
7       for each $(p,v) \in T_e$ {
8         consequent$_e$ = consequent$_e$ $\cap\ p_t = v$
9       }
10      $edgeconstr_{c,e,t}$ = antecedent$_e$ $\rightarrow$ consequent$_e$
11    }
12    antecedent$_{null}$ = $(EV_{c,t} = NULL)$
13    consequent$_{null}$ = $NULL$
14    for each $e \in E_c$ {
15    consequent$_{null}$ = consequent$_{null}$ $\cap$ NOTconsequent$_e$
16    }
17    $edgeconstr_{c,null,t}$ = antecedent$_{null}$ $\rightarrow$ consequent$_{null}$
18  }
19 }
```

Figure 4: Algorithm to generate edge constraints

emit trigger signal *g* as ED_g.

The constraints for value signals are different from those for trigger signals because trigger signals have only instantaneous values. A value signal will keep its previous value until an edge *e* which emits the value signal with a different value is traversed. The equations which express these constraints are produced using the algorithm in Figure 6. In the algorithm, $VSIG$ represents the set of all value signals, V_l represents the set of values for value signal *l*, ED_l is the set of edges which emit value signal *l*, $EDN_{l,v}$ is the set of edges which emit value signal *l* to be all the other values except *v*. Two conditions will set the value of the signal *l* to be *v* at time *t*. First, at least one of the edges emitting the signal with value *v* is traversed at time *t*, which is described in lines 5-9 in the Figure 6; The other condition is that the signal is already set to be *v* at time $t-1$ AND none of those edges emitting it to be other values is triggered at time $t-\delta$. This is described in lines 10- 14 in Figure 6.

Fault Detection Constraints

Additional constraints are required to ensure that the solution generated detects a particular fault. The fault criteria expressed earlier are directly expressed as constraints on the variables associated with the signal involved in a fault. For trigger signals, fault detection is accomplished by forcing a signal definition associated with a fault to occur while the using machine is in the incorrect state. For example, to detect the MTE_{late} fault on the $*short$ signal in Figure 2, the $*short$ signal must be asserted while the HIGHWAY in in the green state. This is accomplished by adding the following constraints.

```
for each $g \in TSIG$ {
    for each $t < TMAX$ {
        **antecedent**$_1$ = $(g_t = 1)$
        **consequent**$_1$ = NULL
        **antecedent**$_0$ = $(g_t = 0)$
        **consequent**$_0$ = NULL
        for each $e \in ED_g$ {
            $c$ is the CFSM containing edge $e$
            **consequent**$_1$ = **consequent**$_1$ $\cup$ $(EV_{c,t-\delta} = e)$
            **consequent**$_0$ = **consequent**$_0$ $\cap$ $(EV_{c,t-\delta} \neq e)$
        }
        $tsigconstr_{c,t,1} =$ **antecedent**$_1$ $\rightarrow$ **consequent**$_1$
        $tsigconstr_{c,t,0} =$ **antecedent**$_0$ $\rightarrow$ **consequent**$_0$
    }
}
```

Figure 5: Algorithm to generate trigger signal constraints

- $SV\_HIGHWAY, 1 = G$
- $TV_{*short,1} = 1$

In this way, the detection criteria of each MTE fault are expressed using 2 constraint equations.

Experimental Results

In order to evaluate our ATPG tool we have used it to detect the MTE faults in the traffic light controller [11], gas station problem [12], and generalized railroad crossing [13]. All the results run on Intel Celeron 566 Processor with 256MB memory and Linux7.1 operating system.

time step		**0**	**1**	**2**	**3**
input signal	*tick*	1	1	1	1
	car	0	1	0	1
	havecar	0	1	1	1
state	highway	green	green	yellow	yellow
	road	red	red	red	red
internal signal	*short*	0	1	0	0

Table 1: MTE Early Fault on *short Signal

Figure 2 presents the network of CFSMs describing a controller for traffic at the intersection of a highway and a country road. Under normal conditions, signal **short* should be triggered when the highway traffic light is yellow. If there is a MTE early fault in the system that triggers the **short* signal when the traffic light of highway is still green, then the traffic light in the highway will be stuck at yellow and the system will halt. Table 1 shows the results of test generation for this fault. Each signal is assigned a value at each time step. Each row describes state or signal in the system, and each column shows the value of these state or signal at each time step. The ATPG tool required 190ms to produce the result.

The gas station problem is a simulation of an automated self-serve gas station [12]. Our version of the gas station consists of three processes: the Customer, the Server, and the

```
for each $l \in VSIG$ {
    for each $t < TMAX$ {
        for each $v \in V_l$ {
            **antecedent** = $(l_t = v)$
            **consequent1** = $NULL$
            for each $e \in ED_l$ {
            $c$ is the CFSM containing edge $e$
            **consequent1** = **consequent1** $\cup$ $(EV_{c,t-\delta} = e)$
            }
            **consequent2** = $(l_{t-1} = v)$
            for each $e \in EDN_{l,v}$ {
            $c$ is the CFSM containing edge $e$
            **consequent2** = **consequent2** $\cap$ $(EV_{c,t-\delta} \neq e)$
            }
            $vsigconstr_{c,v,t} =$ **antecedent** $\rightarrow$
                **(consequent1** $\cup$ **consequent2)**
        }
    }
}
```

Figure 6: Algorithm to generate value signal constraints

time step		**0**	**1**	**2**	**3**	**4**	**5**
input signal	*car*	1	0	0	0	0	0
	tick1	1	1	1	1	1	1
	tick2	1	1	1	1	1	1
	paykey	15	15	15	15	15	15
state	station	idle	wait	wait	pump	pump	pump
	customer	idle	idle	fill	fill	fill	fill
	pump	idle	idle	idle	idle	fill3	idle
internal signal	*pay*	0	1	0	0	0	0
	pump	0	0	0	1	0	0
	pump	5	5	5	5	15	15
	paid	0	0	15	15	15	15
output signal	*fill*	0	0	0	0	0	1

Table 2: MTE Late Fault on Pump Signal

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.

This system contains 6 potential MTE faults associated with the value signal *pump*. Table 2 shows the test generation results for MTE late fault on *pump*. In this case we assume that the initial value of *pump* is 5. If the definition of signal *pump* to value 15 takes more time than the definition of the trigger signal **pump*, *pump* will keep its old value 5 when **pump* is triggered, so edge P3 will be triggered instead of the correct edge P1 in CFSM Pump. Functionally this means that a customer paying for 15 gallons receives 5 gallons of

time step		**0**	**1**	**2**	**3**	**4**	**5**
input signal	**tick1*	1	0	1	0	1	0
	**tick2*	1	0	1	0	1	0
	**tick3*	0	0	1	0	0	1
	**traininP*	0	1	0	1	0	1
	**trainoutI*	1	0	1	0	1	0
state	controller	tEnter	lower	lower	tExit	tExit	tEnter
	gate	up	up	goDown	down	down	down
	train	nHere	nHere	nHere	nHere	nHere	nHere
internal signal	**trainEnter*	0	0	0	0	0	0
	**trainExit*	0	0	0	1	0	0
	**lower*	0	1	0	0	0	0
	**raise*	0	0	0	0	0	1

Table 3: MTE Early Fault on *lower Signal

gas. The ATPG tool required 1760ms to produce the result.

The Generalized Railroad Crossing (GRC) system contains one railroad track protected by a gate and a gate controller. The track is divided into three regions: I(intersection), P(an interval preceding the intersection) and notHere(everywhere else). The gate can be in any of four states: down, up, goingDown, and goingUp. Initially the train is notHere and the gate is in state up. The track is equipped with two sensors: one located at the beginning of the P, triggered when the front of the train enters, and one at the end of the I, triggered when the train completely leaves the intersection. Table 3 shows the test generation result when there is a MTE early fault on signal **lower*. In Table 3, 'tEnter', 'tExit', 'goDown' and 'nHere' represent 'trainEnter', 'trainExit', 'goingDown' and 'notHere' separately. The ATPG tool required 10ms to find a test sequence to detect this fault.

Conclusions

We present an automatic test generation technique for the co-validation of hardware-software systems. We formulate the test generation problem as a set of non-linear constraints on integer variables which collectively describe the space of all system computations. The test generation approach targets the detection of errors in synchronization between concurrent processes which arise from timing faults at communication interfaces. Our future work will investigate a new formulation whose constraints include fewer disjunctive clauses which is a significant source of computational complexity in constraint logic programming.

References

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

[2] S. Arekapudi, F. Xin, J. Peng, and I. G. Harris, "Test pattern generation for timing-induced functional errors in hardware-sofware systems," in *High-Level Design Validation and Testing Workshop*, 2001.

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

[4] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap, "The CLP(R) language and system," *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 3, pp. 339–395, July 1992.

[5] 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.

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

[7] C. Paoli, M.-L. Nivet, and J.-F. Santucci, "Use of constraint solving in order to generate test vectors for behavioral validation," in *High Level Design Validation and Test Workshop*, 2000, pp. 15–20.

[8] 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.

[9] F. Corno, M. Sonze Reorda, G. Squillero, A. Manzone, and A. Pincetti, "Automatic test bench generation for validation of RT-level descriptions: an industrial experience," in *Design Automation and Test in Europe*, 2000, pp. 385–389.

[10] 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.

[11] Palnitkar S., *Verilog HDL*,, Prentice Hall, 1996.

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

[13] Bjormer N., Manna Z., Siopma H. B., and Uribe T. E., "Deductive verification of real-time systems using step," *Theoretical Computer Science*, vol. 253, no. 1, pp. 27–60, 2001.