

An Assertion-based Language for Generating Test Sequences for Complex Temporal Behavior

Pritam Roy¹, Pallab Dasgupta¹, P P Chakrabarti¹

Abstract

This paper addresses the task of stimulus generation for complex temporal behavior of designs. Such stimuli can be used in a variety of cases including simulation at lower level of design hierarchy, synthesis of controllers and post-silicon testing. In this paper we present a language for expressing temporal behavior in the form of formal properties and for annotating various kinds of input constraints to appropriately direct test generation. We present a tool that accepts our language and a circuit as a net-list, and produces sequential test patterns.

1 Introduction

In recent times, there has been a strong focus on *assertion-based validation* (ABV) within the validation flows of leading chip design companies. There are broadly two methodologies for verifying assertions, namely model checking (formal) and simulation based ABV (semi-formal). Current model checking tools do not scale to circuits of large size, thereby limiting the applicability of formal assertion verification to individual component modules of a design. Also, model-checking techniques cannot handle the circuit complexity at the lower levels of design (such as transistor / SPICE level).

While validating designs of large size, the designer often aims to check certain correctness requirements (framed as assertions). If these cannot be verified formally (due to capacity bottlenecks), the designer aims to develop the appropriate stimuli to exercise the scenarios of concern (as a test bench) and then run the simulation with these stimuli for checking the assertions. In recent times several ATPG based techniques [1,2,3,6,7] have been developed for this purpose.

¹ Dept of CSE, Indian Institute of Technology Kharagpur,
Email: {pritam,pallab, ppchak}@cse.iitkgp.ernet.in

In this paper, we present a simple language and a prototype tool for specifying constrained scenarios (as assertions) that can be easily used for generation of stimuli. The main idea is to enable the designer to specify the scenarios of concern in a simple syntax, and then use the proposed tool to generate the stimuli for those scenarios. The generated stimuli can then be used for simulation at lower levels of the design hierarchy, where capacity limitations prevent random / undirected simulation.

The language proposed in this paper is called the *Stimulus Generation Language (SGL)*. The syntax of the proposed language is similar to Linear Temporal Logic (LTL), with two major differences, namely (a) the syntax is much simpler than that of LTL, and (b) it allows the designer to embed various kinds of constraints on the input pattern and the kind of stimuli that is expected. We feel that the latter is an important feature, since LTL does not distinguish between the input and output signals of a module, and hence, determining the triggering stimulus for an LTL property is a very complex problem.

In this paper we also present a tool, called *Stimulus Generation Tool (SGT)* that takes as input a circuit specified as a structural net-list and a property in our language. The tool generates stimuli that serve as a witness for the given property in the given circuit.

2 An Example

Let us take a very simple Vending Machine example. Let us assume the price of Coke is 20 p. We can give only 5 p and 10 p coins. The states denote the internal states of the vending machine, and the state labels indicate the total value of coins that have been fed. The initial state is 00 (the *start state*) and the *target state* is 20. We shall define a stimuli as a sequence $[Q_1 \dots Q_k]$ of test vectors, where each Q_i is a valuation of the inputs to the module.

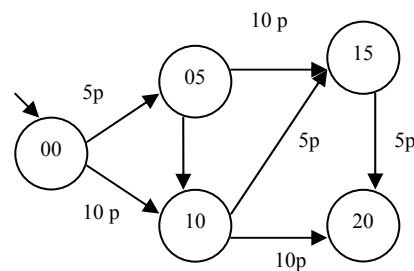


Figure.1. *Vending Machine Example*

Figure.1 shows that there is more *than one path* from the start state to the target state. Suppose we wish to find a way to get the coke by inserting at most 2 coins. We can express this requirement in the proposed logic as:

(true U "20") -clocks 2

One possible stimulus for this property is $[\{10p\}, \{10p\}]$.

Now consider the case when we don't have 10 p coins and still want to get a coke. The query may be expressed in our logic as:

(true U "20") -inexp \neg "10p"

Let us again suppose that we wish to get a coke using coins of a single denomination. This may be expressed as:

(true U "20") -flips 0

One possible stimulus for this property is $[\{5p\}, \{5p\}, \{5p\}, \{5p\}]$.

The proposed language also allows concatenation of the constraint types such as -clocks, -inexp, and -flips.

3 Syntax and Semantics of SGL

Formally, we define a *module* as a tuple, $J = \langle AP, S, I, R, S_0, F \rangle$ where:

- AP is a set of atomic propositions,
- S is a finite set of states,
- I is a finite set of inputs,
- $R: S \times I \rightarrow S$ is the next-state function. Given a state S_i in S, $S_i = R(s_j, \eta)$ is the next state of s_j under input η ,
- S_0 in S is the initial state,
- $F: S \rightarrow 2^{AP}$ is a labeling of states with atomic propositions true in that state.

A *path*, π , in the module is an infinite sequence of state-input pairs $(S_0, \eta_0), (S_1, \eta_1), \dots$, such that for all i , $S_i \in S$ and $S_{i+1} = R(S_i, \eta_i)$. S_0 is called the *start state* of π . Since the module has a finite set of states, one or more states will appear multiple number of times on a path. In other words, a path (as defined here) is an infinite *walk* over the state transition graph.

3.1 Syntax of SGL

The formal syntax of the proposed logic is as follows. Let AP denote the set of state labels (atomic propositions). B denotes the set of Boolean formulas, M denotes the fragment of LTL that we use for assertion specification, and I denotes the set of input constraints. The remaining symbols show the kinds of directives that our language supports to enable constrained stimulus generation.

- $B = p \mid \neg p \mid B \vee B$ where $p \in AP$
- $M = B \mid M \vee M \mid XM \mid B U M$
- $I = i \mid I \vee I$ where $i \in I$
- $IC = (\text{epsilon}) \mid I \mid I;IC$

- IEXP = (epsilon) | - inexp IE
- IE = i | \neg IE | IE & IE | IE \vee IE where $i \in I$
- CCOST = (epsilon) | - clkcost w_1 where $w_1 \in \mathbb{N}$
- FCOST = (epsilon) | - flipcost w_2 where $w_2 \in \mathbb{N}$
- CCONS = (epsilon) | - clocks w_1 where $w_1 \in \mathbb{N}$
- FCONS = (epsilon) | - flips w_2 where $w_2 \in \mathbb{N}$
- CTRL = M IC IEXP
- CONSCTRL = CTRL CCONS FCONS
- SGL = CONSCTRL CCOST FCOST

Cost Function: In SGL, we also support the notion of costs to help us choose between stimuli that serve as different witnesses to the same property. The cost of a path-trace is a weighted sum of w_1 , cost per transition and w_2 , cost per input change. Now we define a cost function as:

- Cost (S_j, p) = 0 , if p is true in S_i ,
- Cost (S_j, p) = ∞ (infinity), if p does not hold in S_i
- Cost ($S_j, \phi_1 \vee \phi_2$) = min (cost (S_j, ϕ_1) , cost (S_j, ϕ_2)) ,
- Cost ($S_j, X \phi$) = cost (S_{j+1}, ϕ) + $w_1 + w_2 * \text{Hamm}(\eta_{j-1}, \eta_j)$ and
- Cost ($S_j, \phi_1 \cup \phi_2$)
= Cost (S_k, ϕ_2) + $w_1 * (k - j) + w_2 * \sum_{i=j}^{k-1} \text{Hamm}(\eta_{i-1}, \eta_i)$

3.2 Semantics of SGL

Prefix (π, S) denotes the sequence of states preceding s in the computation path π . Let us assume $S_j, \pi \models \phi$ indicates formula ϕ holds at state S_j in a computation path π where $\pi = (S_0, \eta_0), (S_1, \eta_1), \dots$ etc. Let us denote π^i a sequence of states in computation path π starting from S_j . Hence we can write, $\pi^i = (S_j, \eta_j), (S_{j+1}, \eta_{j+1}), \dots$

- $S_j, \pi \models \neg \phi$ if $S_j, \pi \not\models \phi$
- $S_j, \pi \models \text{true}$ for all $j \geq 0$
- $S_j, \pi \models p$ iff p in $F(S_j)$
- $S_j, \pi \models \phi_1 \vee \phi_2$ iff $S_j, \pi \models \phi_1$ or $S_j, \pi \models \phi_2$
- $S_j, \pi \models X \phi$ iff $S_{j+1}, \pi \models \phi$
- $S_j, \pi \models \phi_1 \cup \phi_2$ iff in path π , there exists a state S_k , such that for all S in $\text{prefix}(\pi, S_k)$ we have $S^j, \pi \models \phi_1$ and $S_k, \pi \models \phi_2$
- $S_j, \pi \models \phi I_1; I_2$ iff $S_j, \pi \models \phi I_1$ or $S_j, \pi \models \phi I_2$
- $S_j, \pi \models \phi_1 \vee \phi_2 I$ iff $S_j, \pi \models \phi_1 I$ or $S_j, \pi \models \phi_2 I$
- $S_j, \pi \models X \phi$ -inexp IE iff $S_{j+1}, \pi \models \phi$ and input vector η satisfies constraint IE, where $S_{j+1} = R(S_j, \eta, \pi)$
- $S_j, \pi \models \phi_1 \cup \phi_2 I$ -inexp IE iff in path π , there exists a state S_k , such that for all $S_i \in \text{prefix}(\pi, S_k)$ we have $S_i, \pi \models \phi_1 I$ and $S_k, \pi \models \phi_2 I$ and each of η_l satisfies constraint IE and set of changeable inputs (in consecutive transitions) belongs to the set I , where $S_{l+1} = R(S_l, \eta_l, \pi)$

- $S_j, \pi \models \varphi_1 \vee \varphi_2$ I-clocks w_1 iff $S_j, \pi \models \varphi_1$ I-clocks w_1 or $S_j, \pi \models \varphi_2$ I-clocks w_1
- $S_j, \pi \models X \varphi$ I-clocks w_1 iff $w_1 \geq 1$ and $S_{j+1}, \pi \models \varphi$ I-clocks $(w_1 - 1)$
- $S_j, \pi \models \varphi_1 \cup \varphi_2$ I iff in path π , there exists a state S_k , such that for all S_l in $\text{prefix}(\pi, S_k)$ we have $S_l, \pi \models \varphi_1$ I and $S_k, \pi \models \varphi_2$ I and $k \leq (j + w_1)$
- $S_j, \pi \models \varphi_1 \vee \varphi_2$ I-flips w_2 iff $S_j, \pi \models \varphi_1$ I-flips w_2 or $S_j, \pi \models \varphi_2$ I-flips w_2
- $S_j, \pi \models X \varphi$ I-flips w_2 iff $S_{j+1}, \pi \models \varphi$ I and $\text{Hamm}(\eta_{j-1}, \eta_j) \leq w_2$ where $S_{j+1} = R(S_j, \eta_j, \pi)$ and $S_j = R(S_{j-1}, \eta_{j-1}, \pi)$.
- $S_j, \pi \models \varphi_1 \cup \varphi_2$ I-flips w_2 iff in path π^j , there exists a state S_k , such that for all S_l in $\text{prefix}(\pi^j, S_k)$ we have $S_l, \pi^j \models \varphi_1$ I and $S_k, \pi^j \models \varphi_2$ I and for all $j \leq l \leq k$, $\text{Hamm}(\eta_{j-1}, \eta_j) \leq w_2$.
- $S_j, \pi \models \varphi$ I-clkcost w_1 -flipcost w_2 iff $S_j, \pi \models \varphi$ I and minimum cost = for all path π^j [$\min[\text{Cost}(S_j, \varphi, \pi^j)]$]

The following example illustrates a typical module where each transition is labeled by Boolean input vector for which the transition is taken. We will also illustrate the syntax and semantics of SGL through the following example.

Example2: Figure 2 shows a simple module with $S = \{000, 001, 010, 011, 100, 101, 110, 111\}$, $AP = \{O_1, O_2, O_3\}$ and $I = \{I_1, I_2\}$. “000” is the initial state. The set of input vectors enabling a given transition is shown as a Boolean bit string beside the transition. For example, the transition from “000” to “110” is enabled by two input vectors $\eta_1 = (1, 0)$ and $\eta_2 = (1, 1)$, which are represented by bit string “1X”.

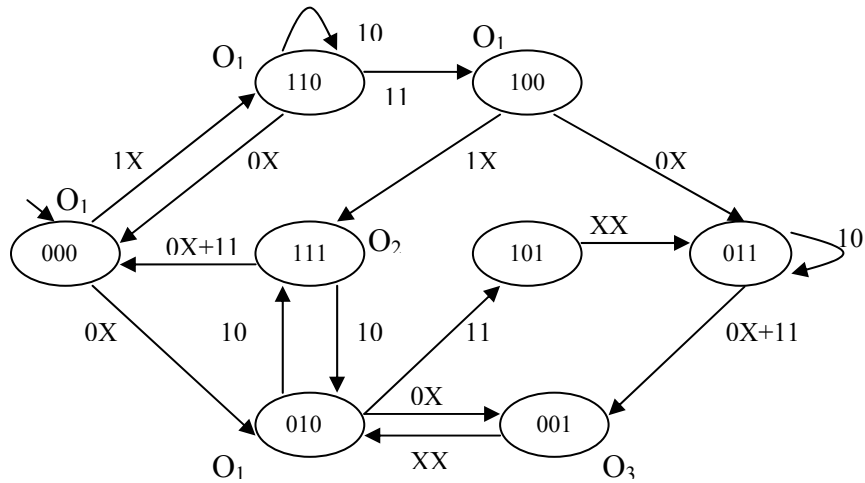


Figure 2: A simple module

Let us consider the following SGL formulas:

```

 $\phi_1 = \text{true} \text{ U } O_2 \text{ -inexp } I_1 \text{ -clocks } 2$ 
 $\phi_2 = \text{true} \text{ U } O_2 \text{ -inexp } I_1 \text{ -clocks } 3$ 

```

Since there does not exist any path from “000” to O_2 (“111”) of length 2 enabled by I_1 , ϕ_1 is false in the start state. But the formula ϕ_2 is *true* in “000” because of the stimulus $\{\{1X\}, \{11\}, \{1X\}\}$.

Suppose we may want to reach O_2 -state and all the previous states in the path are O_1 -state in minimum number of clocks and minimum input flips respectively. The corresponding SGL logic will have the following form:

```

 $\phi_3 = O_1 \text{ U } O_2 \text{ -clkcost } 1 \text{ -flipcost } 0$ 
 $\phi_4 = O_1 \text{ U } O_2 \text{ -clkcost } 0 \text{ -flipcost } 1$ 

```

Both the formulas are true in start state. The stimulus for ϕ_3 is $\{\{0X\}, \{10\}\}$ and the stimulus for ϕ_4 is $\{\{1X\}, \{11\}, \{1X\}\}$.

Sometimes, we are interested to reach a target state without flipping any input variable. We can write this in SGL as:

```

 $\phi_5 = \text{true} \text{ U } O_3 \text{ -flips } 0$ 

```

In the next section we present symbolic BDD-based verification algorithm for SGL verification of modules.

5. Model Checking Algorithms

The SGL Model checking problem may be defined as follows:

Given a Model M and a SGL formula ϕ the SGL Model Checking algorithm determines the set of states S such that for all s in S , $s \models \phi$.

5.1 State Labeling Algorithm

A state s of the Model M is labeled with a SGL formula ϕ if $s \models \phi$. The labeling algorithm we use here is similar in style to the labeling algorithms used by Clarke *et al.* [4,8]. A state s labeled with $\langle f, ic, cl, in \rangle$ means that s holds in f and it can be verified within cl clock transitions and a minimum of in input flips starting from state s with input configuration ic . We now formally describe the labeling schemes for different SGL formulas .

- *Atomic Propositions*: A state s is labeled with $\langle q, \text{true}, 0, 0 \rangle$ iff q is an atomic proposition.
- *$f \vee g$ Formulas*: We label a state s with $f \vee g$ if the state has either of labels f or g . There can be 3 cases.

1) If the state s has label $\langle f, ic_1, cl_1, in_1 \rangle$ and $s \not\models g$ then s is labeled with $\langle f \vee g, ic_1, cl_1, in_1 \rangle$.

2) If the state s has label $\langle g, ic1, cl1, in1 \rangle$ and $s \neq f$ then s is labeled with $\langle f \vee g, ic1, cl1, in1 \rangle$.

3) If state s have labels $\langle f, ic1, cl1, in1 \rangle$ and $\langle g, ic2, cl2, in2 \rangle$ then we have to compute minimum cost = $\min (cl1 * w_1 + in1 * w_2, cl2 * w_1 + in2 * w_2)$.

If former cost is less than latter cost, we copy attribute of f -labeling, otherwise we copy attributes of the g -labeling.

- **X f Formulas:** To handle such formulas, we start from states labeled with f and find the predecessor states satisfying current set of input constraints and label them with $X f$. For example, let there be a transition from state a to state b with input I . Let state b has a label $\langle f, ic, cl, in \rangle$. Then Δi (input bits changed) = $\sum_{i \in IC} I[i] \wedge ic[i]$.

If $\Delta i \leq \text{maxinputchange}$ and $(cl+1) \leq \text{maxclocks}$ then only we can label state a with $\langle Xf, ic', cl+1, in+ \Delta i \rangle$ where $ic'[i] = I[i]$ where $I[i] \neq ic[i]$, otherwise $ic'[i] = ic[i]$

- **f U g Formulas:** In this case we start from the states labeled with g and then do a backward reach-ability satisfying the constraints on the model graph to find all states that can be reach these g labeled states by a path in which each state is labeled with f without violating the constraints.

5.2 Symbolic Procedure for Verifying SGL formula

We have developed a BDD-based symbolic model checking and stimulus generation algorithm for SGL formulas. The algorithm works in the same style as the symbolic model checking algorithm for CTL verification [4,8] but our algorithm also matches input and clock constraints. The verification function *Check* takes the SGL formula to be checked as its argument and returns a OBDD that represents the states of system which satisfy the formula. The procedure *Check* recursively handles the SGL formulas as follows:

- $Check(Xf \text{ - inexp } I \text{ -I Flips } F) = CheckMinX(F, I, Check(f))$
- $Check(Xf \text{ -inexp } I) = CheckMinX(\#input, I, Check(f))$
- $Check(f \text{ U } g \text{ -inexp } I \text{ -clks } C \text{ -flips } F) = CheckMinU(C, F, I, Check(f), Check(g))$

Given the BDD $R(s, i, s')$ for the transition relation, the BDD $I(i)$ for the input constraint I and the BDDs returned by *Check(f)* and *Check(g)* respectively. Again, *FISAD* is a recursive procedure which takes set of states, say $f(s)$ and an integer, n and returns a set of states (in BDD format) which are n input flip away from $f(s)$.

- $\text{CheckMinX}(b, I, \text{Check}(f)) = \text{there exist } s', i, [R(s, i, s') \ \& \ I(i) \ \& \ (\bigvee_{x=0}^b \text{FISAD}(f(s), x))]$
- $\text{CheckMinU}(0, b, I, \text{Check}(f), \text{Check}(g)) = g$
- $\text{CheckMinU}(a, b, I, \text{Check}(f), \text{Check}(g)) = z(s') \vee [f(s') \ \& \ \text{CheckMinX}(b, I, z(s'))]$, where $z(s') = \text{CheckMinU}(a-1, b, I, \text{Check}(f), \text{Check}(g))$

For the *unbounded until* operator, we use usual *fix point* computation

- $\text{CheckMinU}(\infty, b, I, \text{Check}(f), \text{Check}(g)) = \mathbf{1fp} \ z(s') [g(s) \vee [f(s') \ \& \ \text{CheckMinX}(b, I, z(s'))]]$

6. Architecture of the Stimulus Generation Tool:

The tool accepts modules in structural net-list format. The transition relation extracted from a module is stored in a BDD. In order to reduce the effective state space of the modules, we developed abstraction Algorithm, which removes portions of the circuit that do not affect state bits concerning the properties to be verified. We observed that this simple algorithm removed major parts of the data path while retaining the control path.

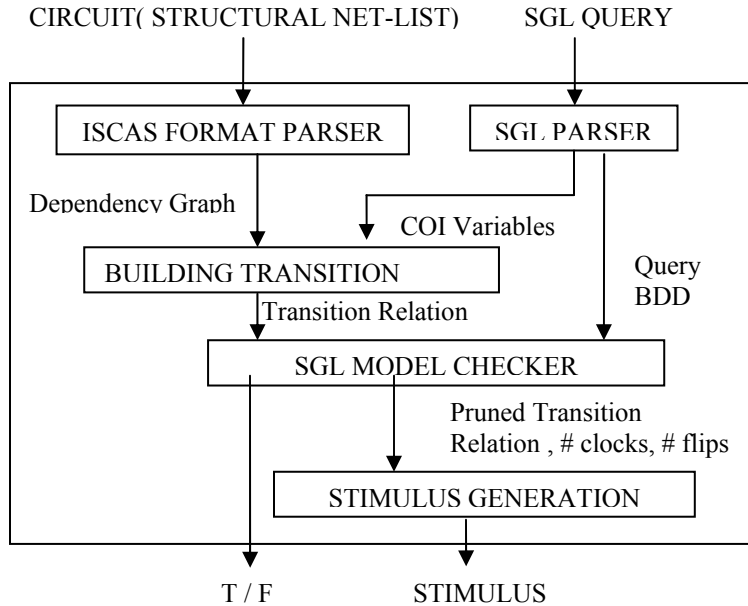


Figure 3. Data Flow of Stimulus Generation Tool

7. Experimental Results

We present experimental results of Stimulus Generation Tool on modules from ISCAS89 Benchmarks. The Stimulus Generation Tool (SGT) was run on a 867 MHz Pentium III machine with 256 MB RAM running Linux 7.0. We used the CUDD BDD [9] package for generating the BDDs.

Ckt name	Input #	Latch #	Gate#	Peak Node #	TR Node #	Pruned TR #	Stimulus Length (# cycles)
S208	10	8	104	6132	1021	760	6
S298	3	14	119	7154	753	390	2
S344	9	15	160	7154	457	400	3
S382	3	21	158	8176	668	609	3
S400	3	21	164	8176	732	555	3
S510	19	6	211	2044	93	54	2
S526n	3	21	194	8176	1903	1717	6
S641	35	19	379	14308	2049	1791	4
S820	18	5	289	4088	548	267	4
S838	34	32	446	15330	4795	4754	14
S953	16	29	395	20440	6395	3895	2

Table 1: Space Requirements of Stimulus Generation Tool

Table 1 and Table 2 shows the time and space required by Stimulus Generation Tool on *ISCAS89* Benchmark circuits respectively. For automatic generation of SGL formulae for these circuits, the input constraints, the start state and the target state are generated *randomly* and written into the corresponding specification file. Peak Node gives the *maximum* BDD node count over total execution time. The next column gives the BDD node count of the transition relation. Pruned Transition Relation Node Count depends on the SGL query.

Ckt name	BUILD TIME (ms)	CHECK TIME (ms)	STIMULUS GENERATION TIME (ms)	TOTAL TIME (ms)	Stimulus Length (# cycles)
S208	4	10	1	13	6
S298	23	2	0	25	2
S344	34	1	1	36	3
S382	78	16	0	93	3
S400	47	17	0	60	3
S510	12	1	0	13	2
S526n	70	44	2	116	6
S641	172	66	2	237	4
S820	15	3	1	19	4
S838	322	60	3	385	14
S953	455	10	2	467	2

Table 2: Time Requirements of Stimulus Generation Tool

The Build time includes building all gates and all latches and the transition relation. In the next column, we show model checking time and building pruned

transition relation for a given formula. The stimulus generation time denotes the time to compute and print the stimulus for the given SGL property on a given circuit. The last column gives the length of the Stimulus. As shown in the Table 2, the stimulus generation for the given SGL property in most of the circuits takes very less time. Hence the stimulus generation logic is very well-suited for automated test pattern generation for sequential circuits.

Acknowledgments

Pallab Dasgupta and P.P.Chakrabarti acknowledge the Dept. of Science & Technology, Govt. of India for partial support of this work.

Bibliography

- [1] Abraham, J.A., Vedula, V.M, Verifying Properties Using Sequential ATPG, *Proceedings of International Test Conference*, October 2002, pp. 194-202
- [2] Bopanna, V. , Rajan, S.P., Tkayama, K. and Fujita, M., “Model Checking Based on Sequential ATPG”, *Computer-Aided Verification*, July 99, pp. 418-429.
- [3] Cheng, K.T., and Kristic, A., “Current Directions in Automatic Test-Generation ”, *IEEE Design and Test*, Vol.32, November1999, pp. 58-64.
- [4] Clarke, E.M., Emerson, E.A., and Sistla, A.P., Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Trans. on Prog. Lang. and Systems*,8(2):244-263, 1986.
- [5] Clarke, E.M., Grumberg, O., and Peled, D.A. (2000), *Model Checking*, MIT Press.
- [6] Hsiao, M. and Jain, J., “Practical use of sequential ATPG for model checking : going the extra mile does pay off ”, *Proceedings Sixth IEEE International High-Level Design Validation and Test Workshop*, 2001, pp. 39-44.
- [7] Keller, B., McCauley, Swenton, J. and Youngs, J., “ATPG in Practical and non-Traditional Applications ”, *Proceedings of the International Test Conference*, October 1998,pp. 632-640.
- [8] McMillan, K.L., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
- [9] Somenzi, F., *CUDD: CU Decision Diagram Package, Release 2.3.0, User's Manual*, Dept. of Electrical and Computer Engineering, University of Colorado, Boulder.