Improving Gate-Level ATPG by Traversing Concurrent EFSMs

Giuseppe Di Guglielmo Franco Fummi Cristina Marconcini Graziano Pravadelli
Dipartimento di Informatica – Università di Verona – Strada le Grazie 15, 37134 Verona, Italy
{diguglielmo, fummi, marconcini, pravadelli}@sci.univr.it, voice +390458027081, fax +390458027068

Abstract

The paper describes an high-level pseudo-deterministic ATPG that explores the DUT state space by exploiting an easy-to-traverse extended FSM model. Testing of hard-to-detect faults is thus improved. Generated test sequences are very effective in detecting both high-level faults and gate-level stuck-at faults. Thus, the reuse of test sequences generated by the proposed ATPG allows to improve the stuck-at fault coverage and to reduce the execution time of commercial gate-level ATPGs.

Keywords: ATPG, EFSM

1 Introduction

Gate-level automatic test pattern generators (ATPGs) represent the state-of-the-art for digital system testing [1, 2, 3, 4]. However, many economical and practical reasons have induced the designers to start automatic test pattern generation at higher abstraction levels [5, 6]. In this way, the verification of complex systems is more tractable and design errors can be early identified and removed, saving time and money. Thus, many high-level ATPGs have been proposed to generate effective test sequences [7, 8, 9, 10, 11, 12, 13]. Generally, high-level ATPGs can be divided in two main categories: random-based and deterministic. The first set adopts simulation-based strategies guided by genetic algorithms or other probabilistic-based techniques [9, 10]. Generally, they rely on high-level fault models or coverage metrics which require to accordingly instrument and simulate an HDL description (e.g., SystemC, VHDL, Verilog, etc.) of the design under test (DUT). These ATPGs are fast, and they allow to quickly achieve an high coverage for easy-to-test designs. However, they tend to generate a large number of test sequences and they unlikely cover corner cases on complex DUTs. On the contrary, deterministic ATPGs exploit mathematical strategies tailored to allow a complete exploration of the DUT state space [7, 8, 11, 12, 13]. Designers of deterministic ATPGs have to select: (a) a model to represent the DUT (e.g., FSM [11], assignment decision diagram [8, 12], BDD [10], etc.); (b) one or more computation algorithms to make decisions during state exploration (e.g., SAT-solving [12], constraint logic programming [13], linear programming [12], etc.); (c) one or more strategies to deterministically reach particular states of the DUT (e.g., learning [14], justification [11], backtracking [8], etc.). Such ATPGs are more effective to cover corner cases than random-based ones, but their execution time tends to increase proportionally to the complexity of the adopted deterministic strategy.

Moreover, deterministic ATPGs can be unable to complete the testing session, when applied to complex systems that lead the DUT model to the explosion of states.

In this paper we propose a pseudo-deterministic high-level ATPG framework which is fast, since it relies on simulation, but also very effective in covering corner cases, since it uniformly explores the state space of the design under test (DUT). The proposed ATPG has been primarily intended for functional verification to early detect design errors. However, it can be exploited to improve gate-level testing too. In fact, the test sequences generated by the proposed ATPG reveal to be very effective to cover gate-level stuck-at faults. Experimental results show how stuck-at coverage achieved by reusing the high-level test sequences is comparable with stuck-at coverage achieved by a state-of-the-art commercial gate-level ATPG. Moreover, in a hierarchical testing context, the commercial ATPG benefits by simulating functional test sequences before applying its generation engines, since this increases fault coverage and decreases execution time.

The main characteristics of the proposed framework are the following.

- The framework accepts VHDL and SystemC behavioral descriptions, on which a set of concurrent extended finite state machines (EFSMs) is automatically identified, one for each process of the DUT. Intuitively, an EFSM is a finite state machine that implicitly memorizes the values of the DUT registers into transitions. Thus, the use of the EFSM model allows a compact representation of the DUT state space that reduces the risk of state explosion typical of more traditional FSMs [15].

- Each EFSM is automatically manipulated to obtain a new EFSM which is more uniformly traversable by exploiting a forward navigation strategy. This allows us to reduce the use of time-consuming backtracking techniques.

- The navigation of concurrent EFSMs is guaranteed by an opportune EFSM scheduling algorithm that aims at maximizing the ATPG capability of exploring the whole state space.

- The quality of test sequence generation is evaluated by using a high-level fault model, the bit coverage, that has been shown to be related to gate-level stuck-at faults [16].

The paper is organized as follows. Section 2 summarizes previous works related to EFSM-based test generation. Section 3 motivates the use of the EFSM model. Section 4 proposes a kind of EFSM that is easy to be traversed. Section 5 shows an overview of...
the proposed high-level ATPG framework. Section 6 and Section 7 describe how the ATPG engine pseudo-deterministically traverses an EFSM for test generation. Section 8 presents the scheduling algorithm defined to explore DUT composed by concurrent EFSMs. Finally, Section 9 reports experimental results and Section 10 is devoted to concluding remarks.

2 Related Work

Some papers, in the literature, propose EFSM-based techniques for high-level test generation [17, 18, 19]. Test generation techniques adopted for FSMs cannot be efficiently reused for EFSMs. In fact, the possibility of traversing a transition of an EFSM may depend not only on values of primary inputs (PIs), like for FSMs, but also on registers whose values have been assigned by previous transitions. Such transitions are note as inconsistent. In [17, 18] different strategies are proposed to remove inconsistencies for reusing FSM-targeted ATPGs. However, the removal of inconsistencies can lead to the state space explosion if the HDL description contains a large number of conditions. A different approach is proposed in [19], where the authors present an orthogonalization process to extract an EFSM model from an HDL description. Then, a stabilization process is presented to improve the traversing of the EFSM. Finally, a breadth first search is used to generate a set of test patterns which cover all the transitions on the stabilized EFSM. The main limitations of this approach are represented by the complexity of the orthogonalization and stabilization process, which may lead to state explosion, and by the breadth-first search, which is not targeted to fault detection. Finally, all previous approaches work on a single EFSM, but this requires either to traverse of the EFSM. The main limitations of this approach are represented by the complexity of the orthogonalization and stabilization process, which may lead to state explosion, and by the breadth-first search, which is not targeted to fault detection. Finally, all previous approaches work on a single EFSM, but this requires either to

3 Computational Model

The first step in designing a digital system consists of formalizing its functionality according to a computational model. Many design methodologies have been proposed in the literature: their computational models fall into three distinct categories [20]: (a) state-oriented, (b) activity-oriented, (c) structure-oriented. The first category represents the system as a set of states and a set of transitions between them triggered by external events (e.g., FSM). Thus, state-oriented models are suited for control systems. The second category is intended to describe the systems as a set of activities related by data and execution dependencies (e.g., dataflow graph). Thus, it is suited for modeling data-dominated systems, where data pass from an activity to the other in a pipelined fashion. The third category is used to describe the system as an interconnection of basic components (e.g., block diagram). Thus, structure-oriented models are less suited than the other categories to specify the functionality, but they are more convenient at lower abstraction levels (e.g., RTL), where component reuse is very common. Finally, many of the characteristics typical of the previously cited categories are merged into heterogeneous models (e.g., program-state machine) to describe different views of complex systems.

In this paper, we represent a digital system as a set of concurrent EFSMs, one for each process of the DUT. In this way, according to the below Def. 1, we capture the main characteristics of state-oriented, activity-oriented and structure-oriented models. In fact, the EFSM is composed of states and transitions, thus it is state-oriented, but each transition is extended with HDL instructions that act on the DUT registers. In this sense, each transition represents a set of activities on data, thus, the EFSM is a data-oriented model too. Finally, concurrency is intended as the possibility that each EFSM of the same DUT changes its state concurrently to the other EFSMs to reflect the concurrent execution of the corresponding processes. Data communication between concurrent EFSMs is guaranteed by the presence of common signals. In this way, structured models can be represented.

Definition 1 An EFSM is defined as a 5-tuple $M = (S, I, O, D, T)$ where: $S$ is a set of states, $I$ is a set of input symbols, $O$ is a set of output symbols, $D$ is a n-dimensional linear space $D_1 \times \ldots \times D_n$, $T$ is a transition relation such that $T : S \times D \times I \rightarrow S \times D \times O$. A generic point in $D$ is described by a n-upla $x = (x_1, \ldots, x_n)$, it models the values of the registers of the DUT. A pair $(s, x) \in S \times D$ is called configuration of $M$.

An operation on $M$ is defined in this way: if $M$ is in a configuration $(s, x)$ and it receives an input $i \in I$, it moves to the configuration $(t, y)$ iff $((s, x, i), (t, y, o)) \in T$ for $o \in O$.

The EFSM differs from the classical FSM, since each transition does not present only a label in the classical form $(i)/o$, but it takes care of the register values too. Transitions are labeled with an enabling function $e$ and an update function $u$ defined as follows.

Definition 2 Given an EFSM $M = (S, I, O, D, T)$, $s \in S, t \in T, i \in I, o \in O$ and the sets $X = \{x|(s, x, i), (t, y, o)\} \in T$ for $y \in D$ and $Y = \{y|(s, x, i), (t, y, o)\} \in T$ for $x \in X$, the enabling and update functions are defined respectively as:

$$e(x, i) = \begin{cases} 1 & \text{if } x \in X; \\ 0 & \text{otherwise}. \end{cases}$$

$$u(x, i) = \begin{cases} (y, o) & \text{if } e(x, i) = 1 \text{ and } ((s, x, i), (t, y, o)) \in T; \\ \text{undef.} & \text{otherwise}. \end{cases}$$

An update function $u(x, i)$ can be applied to a configuration $(s_1, x)$ if there is a transaction $t : s_1 \rightarrow s_2$, labeled $e/t$, such that $e(x, i) = 1$. In this case we say that $t$ can be fired by applying the input $i$.

4 EFSM Manipulation

Many equivalent EFSMs can be generated starting from the same HDL description of a DUT. For example, let us consider the functional description of Fig. 1(a), where in1 and in2 are PIs, out1 and out2 are primary outputs (POs), and reg is an internal register. Fig. 1(b) and Fig. 1(c) show the state transition graphs (STGs) of two EFSMs, respectively $M$ and $M'$, which can be extracted from the code of Fig. 1(a). They are functionally equivalent but the probability of exploring the whole state space navigating $M'$ is more uniformly distributed than navigating $M$.

To explain this concept, let us consider a pseudo-deterministic ATPG that explores the DUT by observing the enabling functions of the corresponding EFSM: (1) the ATPG selects the transition $t$ it wants to traverse (2) it collects information about the PIs involved in the enabling function of $t$; (3) it assigns values to such PIs to
satisfy the enabling function of \( t \); (4) finally, it assigns random values to PIs non involved in the enabling function of \( t \).

Now, let us consider block B11 in Fig. 1(a). What is the probability that such an ATPG generates a test sequence that allows us to enter on B11 by traversing the EFSM of Fig. 1(b)? It is the probability of traversing \( t_3 \) multiplied by the probability of traversing \( t_5 \) by assigning 0 to the input \( \text{in}_1 \). However, \( \text{in}_1 \) is not involved in the enabling function of \( t_5 \), thus the ATPG drives \( \text{in}_1 \) randomly. If \( \text{in}_1 \) is a 32 bit-sized integer, the probability of assign 0 to \( \text{in}_1 \) is \( \frac{1}{2^{32}} \). Thus, it is very unlikely B11 can be reached by pseudo-deterministically traversing this EFSM. On the contrary, the probability that the same ATPG generates a test sequence to enter on block B11 by traversing the EFSM of Fig. 1(c) is the probability of traversing \( t_3^2 \) multiplied by the probability of traversing \( t_5^2 \). In this case, all information required by the ATPG are available on the enabling functions, thus, it can deterministically assign the opportune values to PIs.

In [21], a set of theoretically-based automatic transformations has been proposed to generate a particular kind of EFSM that allows an ATPG to easily explore the state space of the corresponding DUT. Such an EFSM is called semi-stabilized smallest EFSM (S\(^2\)EFSM), since it is composed of few states and it is partially stabilized to remove inconsistencies by following the algorithm proposed in [19]. It presents the following characteristics:

- It is functionally and timing equivalent to the HDL description from which it is extracted, i.e., given an input sequence, the HDL description and the corresponding S\(^2\)EFSM provide the same result at the same time.

- The update functions contain only assignment statements. This implies that all information needed by a deterministic ATPG to traverse the S\(^2\)EFSM resides in the enabling functions. Thus, the probability of firing the transitions of the S\(^2\)EFSM is more uniformly distributed than in an EFSM whose update functions contain conditional statements.

- The S\(^2\)EFSM is partially stabilized to reduce the state explosion problem that may arise when stabilization is performed to remove inconsistent transitions. Only transitions whose probability of being fired is very low are stabilized.

5 EFSM-based ATPG Architecture

The S\(^2\)EFSM model is specially suited to be used with ATPGs that generate test sequences by deterministically activating the enabling functions of the transitions. According to this observation, we propose the pseudo-deterministic functional ATPG framework depicted in Figure 2. It has been implemented in C++/SystemC, however, it accepts both SystemC or VHDL descriptions of the DUT. The framework is composed of two main modules: the DUT-dependent component generator (DCG) and the run-time engine (RTE).

Given a VHDL or SystemC functional description of the DUT, the DCG automatically generates: a SystemC version of the DUT (whenever the original DUT is coded in VHDL, since the simulation engine works on SystemC code only); the concurrent S\(^2\)EFSMs which model the DUT (one for each process); a SystemC description of the constraints that represent the enabling functions of the S\(^2\)EFSMs; and, finally, a SystemC faulty description of the DUT and the related fault list. The faulty description is obtained by automatically inserting saboteurs into the DUT description. Generally, a saboteur is a special component added to the original model [22], whose mission is to alter the value, or timing characteristics, of one or more signals depending on a control line. The way a saboteur perturbs a DUT depends on the adopted fault model. Experimental results reported in Section 9 have been computed by using saboteurs which implement the characteris-
tics of the bit coverage fault model, since it has been shown to be strictly correlated to the gate-level stuck-at fault model [16].

The RTE is a SystemC library which can be linked to the constraint description, and to the faulty and fault-free DUTs to obtain a single executable. It is composed of an $S^2$EFSM navigator, a constraint solver interface, a random engine, and a simulation engine.

The $S^2$EFSM navigator walks across the representation of the $S^2$EFSMs to generate test sequences according to the selected high-level fault model. An external constraint solver is used to generate values for PIs of the DUT which allow to fire the enabling functions of transitions that the $S^2$EFSM navigator wants to traverse. Values for PIs, not involved in the enabling function, are provided by the random engine. Then, the generated test sequences are provided to the simulation engine which compares the behavior of the fault-free and faulty DUTs. Test sequences that highlight discrepancies between the POs of the fault-free and faulty DUTs constitute the final test set.

Next sections describe how the $S^2$EFSMs are traversed by the RTE to generate test sequences.

6 Constraint Generation

The constraint generator automatically creates a SystemC function (eval_func) for each $S^2$EFSM of the DUT. The eval_func allows the RTE to evaluate the enabling functions of the corresponding $S^2$EFSM when these are traversed by the $S^2$EFSM navigator. Each function consists of a single case statement with one alternative for each transition of the corresponding $S^2$EFSM. When the function is invoked by the $S^2$EFSM navigator, the alternative related to the transition to be fired executes as follows:

1. At each simulation cycle, the simulation state is frozen and the values of DUT internal registers are retrieved. Then, the evaluation of the enabling function of the transition to be fired starts.

2. Some conditions of the enabling function can be evaluated without invoking a constraint solver, i.e., those which involve only internal registers and constants. If such conditions are not satisfied, the eval_func return false. Thus, the transition cannot be fired, and the $S^2$EFSM navigator must choose a different one. Otherwise, conditions which involve PIs are extracted from the enabling function, and a constraint solver is called.

3. If the constraint solver provides a solution, then the values returned by the solver are collected to compose the test vector as described in the next section. If the constraint solver fails, false is returned like in step 2.

7 $S^2$EFSM Navigation

The $S^2$EFSM navigator is the heart of the ATPG engine. It exploits the information provided by the enabling functions of the $S^2$EFSM to uniformly move across the transitions of each $S^2$EFSM of the DUT. In this way, the capability of detecting random resistant fault is increased. On the contrary, a random ATPG tends to traverse only transitions whose enabling functions present a high probability of being satisfied by assigning random values to PIs.

Starting from a reset condition, the $S^2$EFSM navigator randomly selects a transition from each $S^2$EFSM according to the scheduling policy described in Section 8. Then, it tries to satisfy the enabling function of each selected transition by exploiting the constraint solver. When it succeeds, the values for PIs provided by the constraint solver are used to generate a test vector. Finally, the simulation engine performs a simulation cycle, by using the generated test vector, to update the internal registers of the DUT, and to check for fault detection. Then, the $S^2$EFSM navigator selects another transition, and the cycle repeats. More formally, given the set $T_{s_i}$ of transitions out-going from a state $s_i$, at step i, the $S^2$EFSM navigator works as follows:

1. Order the $S^2$EFSMs according to the scheduling policy reported in Section 8.

2. For each $S^2$EFSMs:
   (a) Randomly choose a transition $t_{s_i} \in T_{s_i}$.
   (b) Call the eval_func described in Section 6 to check if the enabling function $e$ of $t_{s_i}$ is satisfiable, i.e., if it can be fired by assigning opportune values to inputs involved in $e$. For example, let $x$ be an input of the $S^2$EFSMs, the enabling function $x = 0$ can be fired by assigning 0 to $x$. On the contrary, if $x$ is an internal register, the satisfiability of the enabling function depends on the previous assignment to $x$, i.e., on the current configuration of the $S^2$EFSMs. This step is performed by invoking a constraint solver by means of the constraint solver interface module. Note that, for DUT composed of more than one $S^2$EFSM, the transitions selected at step (a) may require to assign conflicting values to the same PIs to be concurrently fired. In this case, according to the scheduling policy reported in Section 8, the $S^2$EFSM with the highest priority wins, while the others must consider such PIs as internal registers whose value cannot be changed.
   (c) Assign to PIs involved in $e$ the values provided by the constraint solver, if $e$ is satisfiable. Otherwise, remove $t_{s_i}$ from $T_{s_i}$ and come back to step 2.

3. Generate random values for PIs not involved in the enabling functions of the selected transitions. To accomplish this task, the random engine is invoked.

4. Invoke the simulation engine to simulate the test vector so obtained, move across the selected transitions, and come back to step 2 to generate the next test vector.

The $S^2$EFSM navigator resets the state of $S^2$EFSMs periodically according to a user parameter, and it finally stops when the target fault coverage has been reached or the maximum allowed computation time has been expired.

8 Multi-Process Scheduling

A scheduling algorithm is required to sort the $S^2$EFSMs of a multi-process DUT, when the pseudo-deterministic navigator presented in Section 7 is adopted. In particular, two reasons induce us to introduce a scheduling algorithm.
A first motivation is due to the fact that the same PIs may be involved in the enabling functions of the transitions of two or more $S^2$EFSMs. For example, let us consider a DUT composed of two $S^2$EFSMs, $M$ and $N$. Moreover, let us suppose that at a certain simulation cycle, the $S^2$EFSM navigator selects the transition $t^M$ from $M$ and the transitions $t^N$ from $N$. The enabling functions of $t^M$ and $t^N$ can be compatible or conflicting. In the first case, there exist a value assignment to PIs that satisfies the enabling functions of both $t^M$ and $t^N$. Thus, PIs can be deterministically fixed to fire both $t^M$ and $t^N$. In the second case, such an assignment does not exist. Thus, $t^M$ and $t^N$ cannot be fired concurrently, and one of them must be discarded and substituted with a different transition to remove the conflict. In this case, the scheduling algorithm is used to decide the priority of each $S^2$EFSMs for fixing PIs. When a conflict happens, the transition to be substituted are selected starting from the $S^2$EFSMs with the lowest priority.

A second reason that motivates the use of a scheduling algorithm depends on the fact that not all $S^2$EFSMs must be triggered at each simulation cycle. Think, for example, to an $S^2$EFSM corresponding to a process whose sensitivity list’s signals remain unchanged.

According to the previous considerations, a priority-based scheduling algorithm has been implemented. In this way, at each simulation cycle, the $S^2$EFSM navigator generates the test vector by deterministically fixing the PIs in order to fire the transition of the highest-priority $S^2$EFSM. Then, PIs not involved in such a transition, are deterministically assigned according to the transition of the next-priority $S^2$EFSM, and so on, until all PIs are fixed.

To implement such a policy, the scheduler relies on a two-level queue mechanism without feedback. The queue with the highest priority permanently includes $S^2$EFSM extracted from clock-sensitive processes. Such $S^2$EFSMs are simulated at each clock cycle. The second queue permanently includes $S^2$EFSMs extracted from asynchronous processes that are triggered by signals modified by the processes of the first queue. Within each queue, the $S^2$EFSMs are sorted by assigning a dynamic priority computed as the sum of a constant value, $F$, and an aging factor $A$.

The value $F$ assigned to each $S^2$EFSM is inversely proportional to the number of PIs included in its enabling functions. Larger is the number of PIs involved in the enabling functions, lower is the value $F$ assigned to the $S^2$EFSM, and later the $S^2$EFSM is navigated. In this way, the DUT exploration is more uniformly distributed. In fact, if an $S^2$EFSM, which involves many PIs on the transition-to-be-fired, is scheduled first, its decision on PIs values definitely constraint the behavior of the remaining $S^2$EFSMs. This may cause an incomplete exploration of the DUT. Analogously, the aging factor has been introduced to avoid that the behavior of low-priority $S^2$EFSMs is always constrained by decisions taken by high-priority $S^2$EFSMs. Initially, the value $A$ is 0 for all $S^2$EFSMs. Then, each time an $S^2$EFSM is forced to discard and substitute the transition to be fired (because the values assigned to PIs by an higher-priority $S^2$EFSM do not satisfy its enabling function) the aging factor of $S^2$EFSM is incremented of a constant quantity. On the contrary, the aging factor is reset to 0 after the $S^2$EFSM reaches the highest priority. Thus, sooner or later, every $S^2$EFSM becomes the highest-priority one.

9 Experimental Results

The efficiency of the proposed ATPG framework has been evaluated by using the benchmarks described in the first part of Table 1, where columns report the number of primary inputs (PIs), primary outputs (POs), flip-flops (FFs) and gates (Gates). Such benchmarks have been selected because they present different characteristics which allow us to analyze and confirm the effectiveness of the ATPG framework. $b04$, $b09$ have been selected from the well known ITC-99 benchmarks suite [6]. $b11m$ is a modified version of $b11$, included in the same suite, created by introducing a delay on some paths to make it harder to be traversed. The original HDL descriptions of $b04$, $b09$ and $b11m$ contain a high number of nested conditions on signals and registers of different size. $ex1$, $b00$ and $b00z$ are internal benchmarks, while $fr$ is a real industrial case, i.e., it is a module of a face recognition system. Finally, $dxl$ is the controller of the well known RISC processor.

The effectiveness of the proposed EFSM-based ATPG has been evaluated by comparing it to a genetic algorithm-based high-level ATPG and to a commercial gate-level ATPG.

9.1 Genetic Algorithm vs. EFSM based ATPG

The second part of Table 1 reports the comparison between the performance of a genetic algorithm-based high-level ATPG (GA-ATPG), which outperforms a pure random-based ATPG, and the performance of the proposed EFSM-based ATPG (EFSM-ATPG). In particular, the Table shows the number of bit coverage faults (B.C.), the fault coverage (FC%), the statement coverage (SC%), and the test generation time ($T$ (sec.)) by using respectively the GA-ATPG and the EFSM-ATPG. The time to obtain the $S^2$EFSM from the HDL code is negligible with respect to the test generation time (few seconds). It can be observed that the EFSM-ATPG outperforms the GA-ATPG. The very low fault coverage achieved by the GA-ATPG for $b00$ and $fr$ is due to the presence of a transition out-going from the initial state, whose enabling function has an infinitesimal probability of being fired by randomly fixing the values of PIs. On the contrary, the EFSM-ATPG efficiently exploits the constraint solver to generate the opportune PIs values by exploring the $S^2$EFSM models. This sensibly increases the achieved fault coverage for all benchmarks.

9.2 Gate-level vs. EFSM based ATPG

Table 2 reports the comparison between the EFSM-ATPG and a state-of-the-art commercial gate-level ATPG. Columns report: the number of stuck-at faults ($F#$) the stuck-at fault coverage (FC%) and the execution time ($T$ (sec.)) by using the gate-level ATPG, respectively, in ATPG mode (ATPG), simulation mode (SIM), and incremental ATPG mode (SIM+ATPG). In ATPG mode, test vectors are automatically generated by exploiting the internal engines of the gate-level ATPG. In simulation mode, we force the gate-level ATPG to simulate only test vectors generated by the EFSM-ATPG to cover bit coverage faults. Finally, in incremental mode, the gate-level ATPG simulates the EFSM-ATPG test vectors and then it tries to increase the fault coverage by exploiting its internal
ATPG engines. The Table shows that the stuck-at fault coverage achieved by reusing the test vectors generated by the high-level EFSM-ATPG is comparable with the one achieved by the gate-level ATPG mode. Moreover, when the gate-level ATPG is used in the incremental mode, it takes a great benefit by simulating test sequences generated by the high-level EFSM-ATPG. In particular, the final fault coverage is always greater than or equal to the one achieved by the ATPG mode. However, by comparing the execution times, we observe that the incremental mode evidently provides better performance than the ATPG mode; we save on average 42% of time.

### 10 Concluding Remarks

We presented an high-level ATPG framework which exploits the EFSM model and a constraint solver to pseudo-deterministically traverse the state space of the DUT. It extracts, from each process of the DUT, a particular kind of EFSM (S²EFSM) that is easy-to-be-traversed. A scheduling algorithm has been proposed to allow a fair exploration of the DUT, by giving to each S²EFSM the possibility of deterministically fixing PIs to reach the desired destination state. The quality of generated test vectors has been compared, at high-level, with a genetic algorithm-based ATPG, and at gate-level, with a commercial ATPG. The EFSM-based ATPG outperforms the genetic ATPG, while it provides comparable stuck-at fault coverage with respect to the commercial ATPG. However, the gate-level ATPG performance greatly benefits by reusing the test vectors generated by the EFSM-based ATPG. In particular, stuck-at fault coverage increases for the half of benchmarks and execution time is decreased on average of 42%.

### References


