A Framework for Testing Core-Based Systems-on-a-Chip [*]

Srivaths Ravi[†], Ganesh Lakshminarayana[‡], and Niraj K. Jha[†]

[†]Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

[‡]C&C Research Labs
NEC USA, Inc., Princeton, NJ 08536

Abstract

Available techniques for testing core-based systems-on-a-chip (SOCs) do not provide a systematic means for synthesising low-overhead test architectures and compact test solutions. In this paper, we provide a comprehensive framework that generates low-overhead compact test solutions for SOCs. First, we develop a common ground for addressing issues such as core test requirements, core access and test hardware additions. For this purpose, we introduce finite-state automata for modeling tests, transparency modes and test hardware behavior. In many cases, the tests repeat a basic set of test actions for different test data which can again be modeled using finite-state automata. While earlier work can derive a single symbolic test for a module in a register-transfer level (RTL) circuit as a finite-state automaton, this work extends the methodology to the system level, and, additionally contributes a satisfiability-based solution to the problem of applying a sequence of tests phased in time. This problem is known to be a bottleneck in testability analysis not only at the system level, but also at the RTL. Experimental results show that the system-level average area overhead for making SOCs testable with our method is only 4.4%, while achieving an average test application time reduction of 78.5% over recent approaches. At the same time, it provides 100% test coverage of the precomputed test sets/sequences of the embedded cores.

1 Introduction

Embedded cores are being increasingly used to provide SOC solutions to complex integrated circuit design problems. Synthesis of low-overhead test architectures and compact test sets for these SOCs is of critical importance [1, 2].

Traditional approaches for testing SOCs rely on variants of boundary scan or test bus to provide test access to the embedded cores [2, 3, 4]. However, the area and delay overheads of such methods are usually high, as is the test application time for boundary scan based methods. Another approach is to use existing functionality for test access of embedded cores [5, 6, 7]. Even though such functional access approaches reduce overheads, they are usually not flexible enough to handle embedded cores made testable using diverse test methodologies such as scan, built-in self-test (BIST), sequential test generation, *etc.*

In this paper, we present a comprehensive technique for testing core-based SOCs. Our solution focuses on effectively reducing the *test application time*, in addition to reducing the test overheads, while providing complete test coverage of the test sets/sequences of the embedded cores. The key strengths of our work are as follows.

- It is able to use individual cores to transfer test data to and from the core under test in a much more *aggressive* and *effective* manner than has been done in previous work. To provide such a functional test access, it allows complex core transparency modes, where core transparency can be loosely defined as the ability of the core to propagate test data from its inputs to its outputs (more formal definition is given later). Consider, for example, a situation where we need to transfer data from an 8-bit input of a core to its 16-bit output. Our technique can make use of the fact that the core might provide a mechanism to compose the output vector from two time-separated input vectors, whereas other techniques [6, 7] would consider the core incapable of executing the required data transfer and solve the problem by adding design-for-test (DFT) hardware such as test multiplexers.
- It provides controllability and observability to cores on an as-needed basis: past work [6, 7] requires all the inputs (outputs) of a core to be simultaneously, though indirectly, controllable (observable). This can lead to significant area overheads, especially if the test scheme for the core does not require all its inputs (outputs) to be simultaneously controllable (observable).
- It explicitly accounts for sequential test sequence requirements of cores: core test strategies apply a single symbolic test (a set of test actions) to a core multiple times with different test data at pre-defined time-instants. As opposed to the existing expensive practice of meeting these test requirements with extra test hardware, our technique tries to overlap multiple tests in time to realize a low-cost solution, whenever possible. It requires SOC-level test insertion only as a last resort. It also effectively models diverse core test strategies like scan and BIST.

The paper is organized as follows. Section 2 introduces transparency and test models through examples. Section 3 presents the SOC testing algorithm. Section 4 gives the experimental results and Section 5 the conclusions.

2 Transparency and Test Models

In this section, we present our models for different aspects of core-based testing. First, we introduce a novel way of modeling the *transparency* of a core using finite-state automata. Next, we model diverse test requirements for a core under test. Lastly, we illustrate a compact and efficient test architecture derived for an SOC with the help of a test strategy (Section 3) employing these models.

2.1 Transparency of a core

We first formalize the notion of transparency used in our work as follows.

Definition 1: Transparency *of a core is a collection C of non-deterministic finite-state automata [8] T_I^J that allows test data to propagate from a set of its inputs I to a set of its outputs J.*

The advantages of using a finite-state automaton (FSA) to describe tranparency are manifold. First, we can compactly describe

[*]Acknowledgments: This work was supported by NSF under Grant No. MIP-9729441.

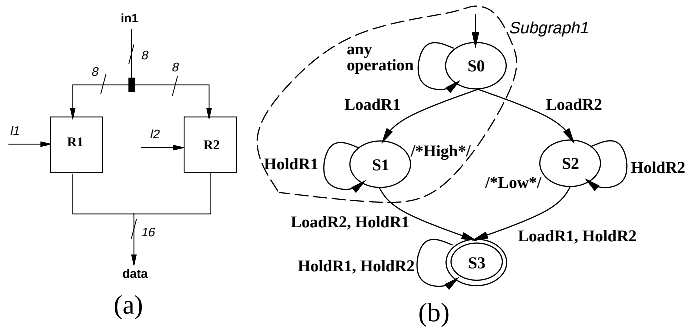


Figure 1: (a) An RTL element, and (b) transparency FSA T_{in1}^{data}

temporally-separated circuit events. For example, if we examine the transparency of a register $R1$, we can deduce that the element (at some time instant) can propagate data at its input upon a $LoadR1$ operation. Subsequently, the data is available at its output for as many cycles as allowed by the $HoldR1$ operation. Subgraph1 in Figure 1(b) compactly captures this behavior (note that $S1$ would need to be changed to an accept state for this purpose).

We can also use an FSA to capture spatial aggregation of test data. Consider, for example, the two 8-bit registers shown in Figure 1(a), whose outputs aggregate to form a 16-bit bus $data$. The transparency behavior for the $in1/data$ pair is given by the FSA, T_{in1}^{data}, shown in Figure 1(b). Start states are depicted by an incoming arrow and accept states by a double ellipse. Observe that the annotation along the states and edges gives us the information necessary to compose our objective in time and space. Suppose that we require a value v at $data$ to be justified from $in1$ in the shortest time possible, assuming no other external constraints on the control signals. The shortest-path solutions directly available from Figure 1(b) are (a) $S0$ to $S3$ via $S1$, and (b) $S0$ to $S3$ via $S2$. Both solutions take two time-steps. In (a), setting of the higher (lower) 8 bits of $data$ is achieved in the first (second) time-step, while in (b), it is the reverse (indicated by the annotation to states $S1$ and $S2$, respectively).

In this way, we can represent the transparency behavior of an RTL element in a compact manner using an FSA. Since a core can be viewed as an interconnection of RTL elements, we can clearly extend the transparency behavior concept to the core-level. For this purpose, we first pre-compute the transparency behavior for the different RTL elements. We next perform RTL symbolic justification and propagation analysis to compose a transparency behavior for the core outputs in terms of the core inputs. Exploiting the equivalence of regular expressions and FSA, we tailored an existing regular expression based justification and propagation framework [9] for this purpose. Modeling of core-level transparency behavior using an FSA is analyzed in the following example.

Example 1: Consider the core $DG1$ shown in Figure 2. It is a part of an SOC used as a data address generator. The core consists of a 16-bit input $Data$, a reset input R, a test input T and a 16-bit output $Generate$. The core has an arithmetic logic unit, ALU, and a counter, CTR, which facilitate both logical and arithmetic modifications to the input data.

Let us analyze the transparency behavior of $DG1$ as shown in Figure 3. The FSA has one start state $S0'$ and six accept states $S0$, $S4, \cdots, S8$. States $S0'$ and $S0$ have the property that incoming transitions to these states preserve the previously held values in the different register elements. Therefore, it follows from Figure 3 that values generated at an accept state (other than $S0$) are always available for one extra time-step through a transition to $S0$. This is useful when we use $DG1$ to feed cores that require the same test data for an extra cycle. The edges in the FSA are annotated with $input$ labels that denote the values required at the primary inputs, while states are annotated

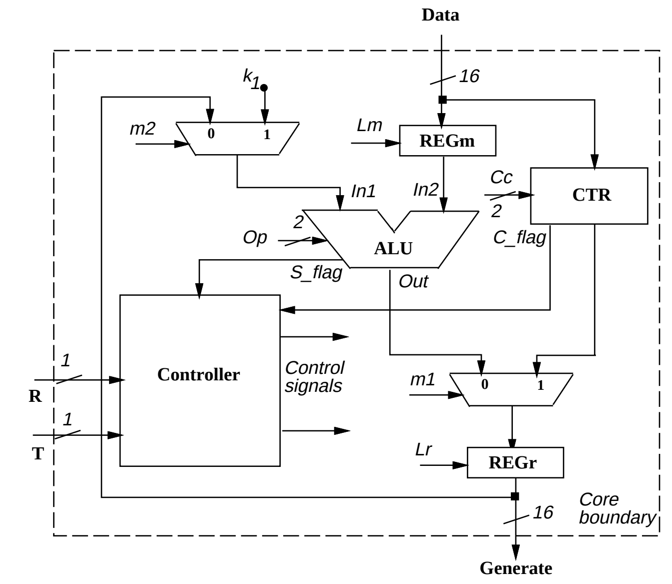


Figure 2: RTL block diagram of core $DG1$

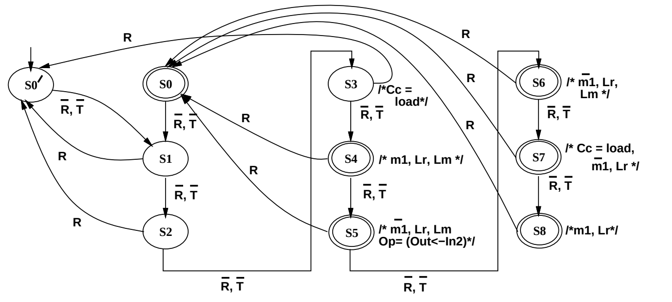


Figure 3: Transparency behavior for $DG1$

with the operations performed in that cycle. For example, the transition from state $S2$ to $S3$ is labeled with $\overline{R}, \overline{T}$, which indicates that an input of $R = 0, T = 0$ when applied to $DG1$ in state $S2$ transfers $DG1$ to state $S3$. In state $S3$, CTR loads its input (as given by the annotation $Cc = load$ to state $S3$). Observe that the FSA is only partially specified, thereby giving only the necessary information required for transparency analysis.

Consider the problem of propagating a test data sequence $< v1, v2 >$ from $Data$ to $Generate$. From Figure 3, we can see that the path $S0'$ to $S5$ (via $S1$, $S2$, $S3$ and $S4$) in the FSA has two accept states, $S4$ and $S5$. If we examine the path $S0'$ to $S4$, we need $v1$ at $Data$ at time $t = 2$ for providing $v1$ at $Generate$ at time $t = 4$. Likewise, path $S0'$ to $S5$ propagates $v2$ at $Data$ at time $t = 3$ to $Generate$ at time $t = 5$. From the annotations to the edges, we can see that the data transfers are activated from the start state by the following input sequence at R, T: 00, followed by 00, 00, 00 and 00. In this way, we can use an FSA to effectively propagate a test data sequence and also determine the additional constraints (*e.g.*, R, T here) that facilitate this transfer. ■

2.2 Core test requirements

Diverse strategies adopted by different core vendors to test a core create a range of controllability/observability objectives for an SOC. For example, high-level symbolic test generation techniques [9] for RTL circuits have controllability and observability requirements only at some specific time-instants (don't cares otherwise). Test strategies such as scan, BIST, sequential test generation, *etc.*, place similar cycle-by-cycle requirements. Consequently, an SOC test framework must be flexible enough to encapsulate different specifications of test sets and also provide a common ground for systematic analysis. In the following example, we illustrate how an FSA can provide a convenient and compact representation to the given core test requirements.

Example 2: Consider the core TLU in the SOC *Sys_Gen* shown in Figure 4 with a test input *Test* that serves to enforce a scan test strategy imposed by the core vendor on TLU. Suppose the scan implementation of TLU has 16 parallel scan chains, each of length 4 (*i.e.*, having 4 flip-flops on its path), running from *Lo* to *Out*. When *Test* = 1, the core scans in (out) the input (output) at *Lo* (*Out*). Since the scan-chains are of length 4, a sequence of four 16-bit state vectors must be scanned in with *Test* as 1 before applying the required input test vector (with *Test* = 0). Suppose the core vendor provided a set of V vectors in the combinational test set to test the scan implementation of TLU. Testing this core in isolation, therefore, consumes $V*(4+1)+3$ clock cycles.

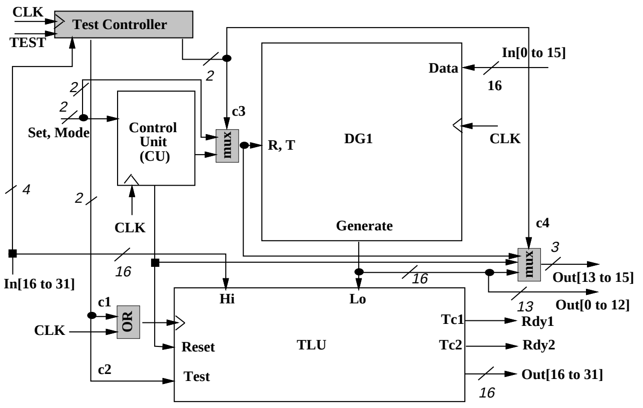


Figure 4: SOC *Sys_Gen* enhanced by our testing scheme

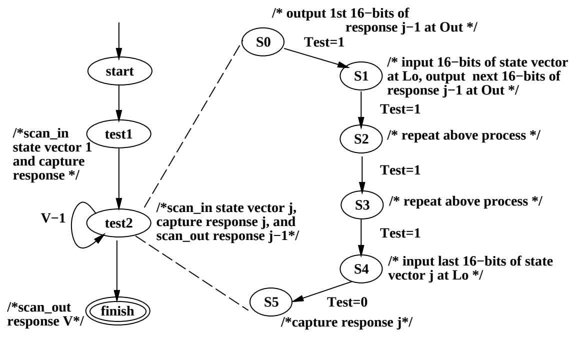
The above test requirements are modeled by the FSA shown in Figures 5(a) and (b). Figure 5(a) models a general scan test schedule as an FSA consisting of a start state *start*, an accept state *finish* and two other states *test*1 and *test*2. Each state represents a high-level granularity of test actions (as represented by the annotations to the states), and, in turn, can be decomposed into an FSA. For example, the steady-state scan actions of applying the test vector after scanning in the state vector and scanning out the previous response (represented by state *test*2 in Figure 5(a)) can be decomposed into the FSA shown in Figure 5(b). This decomposition is shown in the context of the scan test requirements of the core TLU. Scanning one test pattern into TLU requires four cycles of shifting from the input with *Test* = 1 (indicated by states *S*0 to *S*4). Simultaneously, we also scan out the stored test response. Thereafter, we can apply the input test vector by setting *Test* = 0 ($S4 \rightarrow S5$ transition). If there are V test vectors, this FSA is executed $V-1$ times accounting for the self-loop at state *test*2. ■

The FSA to model sequential test sequences and BIST can be obtained in a similar fashion. For example, we can use a sequential test generator such as HITEC [10] to generate a sequence of test vectors for the gate-level implementation of core *DG*1. We can then model this test schedule by the FSA shown in Figure 6(a). Similarly, we can model a BIST test scheme for a memory module that has a 1-bit *Test_Start* input and a 1-bit *Test_Flag* output by the FSA shown in Figure 6(b). In this way, FSA representations of basic test schedules provide a common ground for creating a systematic framework for their analysis (Section 3). This, in turn, leads to a compact and low-cost test architecture design discussed next.

2.3 System test architecture

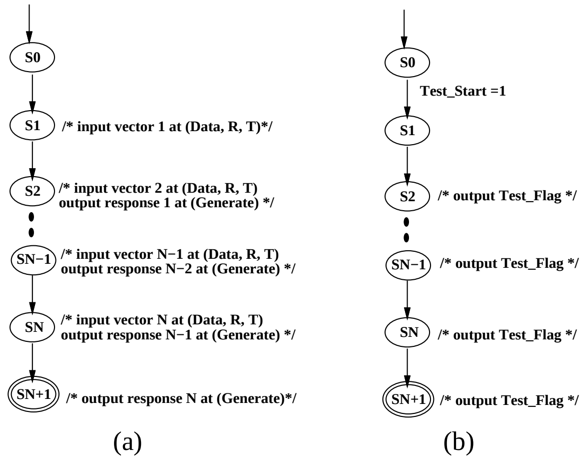
In this section, we first introduce an example system test architecture generated for an SOC, and then quantitatively analyze it for possible test application time savings in comparison with existing techniques.

Example 3: Consider the SOC, *Sys_Gen*, shown in Figure 4 once again. The blocks shaded in grey indicate the additional hardware that



/* output 1st 16−bits of response j−1 at Out */
S0
Test=1
start
S1
/* input 16−bits of state vector at Lo, output next 16−bits of response j−1 at Out */
Test=1
/*scan_in state vector 1 and capture response */
test1
S2
/* repeat above process */
Test=1
V−1
test2
/*scan_in state vector j, capture response j, and scan_out response j−1*/
S3
/* repeat above process */
Test=1
S4
/* input last 16−bits of state vector j at Lo */
/*scan_out response V*/
finish
S5
Test=0
/*capture response j*/
(a)
(b)

Figure 5: (a) An FSA depicting a general scan test schedule, and (b) partial decomposition to meet TLU's scan test requirements



S0
S1
/* input vector 1 at (Data, R, T)*/
S2
/* input vector 2 at (Data, R, T) output response 1 at (Generate) */
SN−1
/* input vector N−1 at (Data, R, T) output response N−2 at (Generate) */
SN
/* input vector N at (Data, R, T) output response N−1 at (Generate) */
SN+1
/* output response N at (Generate)*/
S0
Test_Start =1
S1
S2
/* output Test_Flag */
SN−1
/* output Test_Flag */
SN
/* output Test_Flag */
SN+1
/* output Test_Flag */
(a)
(b)

Figure 6: FSA representing test schedules for (a) sequential test generation for *DG*1, and (b) BIST of a memory module

form a part of the low-overhead test architecture obtained by using the algorithm proposed in Section 3. *Sys_Gen* manipulates a 32-bit input *In* using two main computational blocks, *DG*1 and TLU, to generate a 32-bit output *Out* and two 1-bit outputs *Rdy*1 and *Rdy*2. A control unit *CU* sequences the activity in *DG*1 and TLU based on two system inputs, *Set* and *Mode*. Assuming that separate test sequences are provided for testing *CU* and *DG*1 (when they are standalone), and a combinational test set for the scan implementation of TLU, we will evaluate the test architecture derived to provide testability to the different blocks in *Sys_Gen*. The test controller on the chip is a simple finite-state machine that loads a set of input vectors (when *Test* = 1) through the existing system inputs *In*[16 *to* 19]. The test controller feeds control inputs *c*1, *c*2, *c*3 and *c*4 of the different test hardware shown.

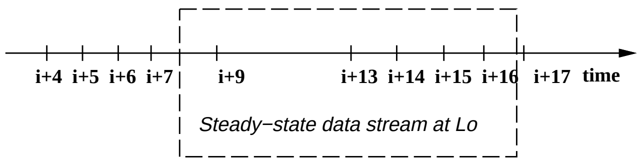


Figure 7: Steady-state data stream at *Lo* for testing TLU

Let us compute the test application time to test TLU under this architecture. Figure 7 shows the steady-state flow of test data at *Lo* from *In*[0 *to* 15] using the transparency behavior of *DG*1. Suppose *DG*1 allows variable-latency transparency. Specifically, assume that the first vector can propagate through *DG*1 in four cycles, but subsequent vectors take only one extra cycle to propagate through it because of a pipeline in it. The scan action described by the FSA in Figure 5 is realized by the window shown in Figure 7. Specifically, four 16-bit vectors are scanned in at *Lo* through *DG*1 at cycles $i+9$, $i+13$, $i+14$ and $i+15$. Since the state of *TLU* must be preserved between cycles $i+9$ and $i+13$, the test controller sets $c1 = 1$ within this period to gate the clock of *TLU* (and thus preserve its state). The circuit response is captured at cycle $i+16$, and scan-out takes four cycles starting at $i+17$. However, since scan-in of a new state vector and scan-out of the previous captured response can occur simultaneously in the window shown, it takes eight cycles per test vector to test *TLU*.

We next compare our scheme with the ones presented in [6, 7] which only allow constant-latency transparency. In other words, under their scheme, *DG*1 can only feed the desired 16-bit vectors to *Lo* every four clock cycles (the clock needs to be gated here as well to preserve the state when necessary). Thus, in the steady state, it takes 16 cycles to scan-in the desired state into *TLU* and four more to feed it the desired test vector, for a total of 20 cycles (scan-out can take place in parallel with scan-in as usual). This means that our scheme results in a test application time speed-up of 2.5X for testing *TLU*. ■

In the next example, we will illustrate how our methodology can also help lower area and delay overheads.

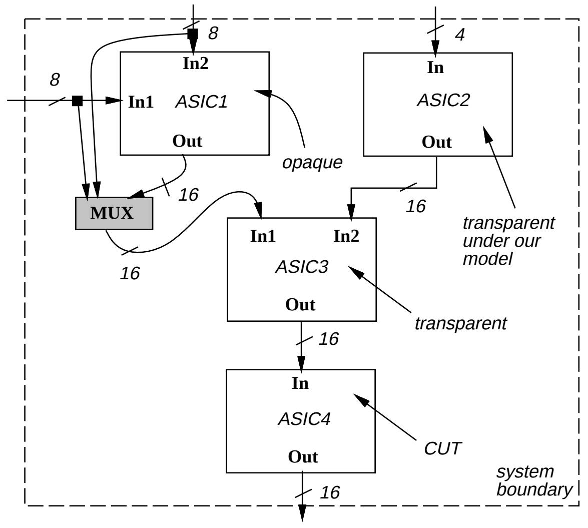


Figure 8: An SOC with an unnecessary test enhancement

Example 4: Figure 8 depicts the main components of an SOC called *SysProc* (ignore the shaded mux temporarily and assume that *ASIC*1.*Out* is connected directly to *ASIC*3.*In*1). Consider the objective of testing *ASIC*4 with a given test sequence at its input *In*, and observing the resulting test responses at its output *Out*. The transparency characteristics of the different cores, which are significant for the testability of *ASIC*4, are as follows: *ASIC*1 is opaque (empty transparency set), while cores *ASIC*2 and *ASIC*3 are transparent. The transparency of *ASIC*2 is significantly different since it takes 4-bit inputs at time-instants 1, 2, 4 and 5 to compose a 16-bit input at time-instant 6. This is depicted by the FSA shown in Figure 9. The testability schemes proposed in [6, 7] cannot model this transparency. Consequently, the only option for those schemes is to provide test data at *ASIC*3.*In*1 through additional test hardware. This is done by adding the shaded test hardware shown in Figure 8 for this purpose, causing area and delay overheads.

Subsequent testability analysis by our algorithm in Section 3 exploits the transparency of *ASIC*2 and determines *ASIC*4 to be testable. Hence, no additional test hardware is necessary for test data access at *ASIC*4.*In* from the system inputs, leading to savings in area and delay

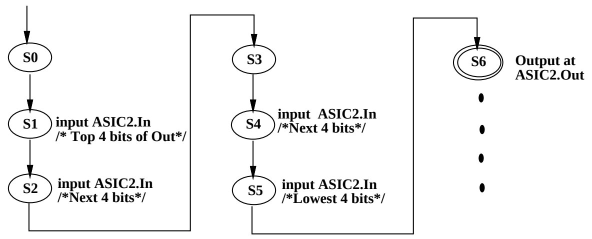


Figure 9: Partial FSA representing transparency behavior of *ASIC*2

overheads. This case study clearly suggests that better test and transparency models are crucial in the development of low-overhead SOC test solutions. ■

3 The SOC Testing Algorithm

In this section, we detail the algorithmic aspects of our methodology. Our algorithm takes as its input a system of cores, their connectivity and test requirements, and outputs a low-overhead test architecture and a test schedule that facilitate its testability. In the process, it follows the steps outlined below.

- The first step in the algorithm is to model the transparency and test requirements of the individual cores (Section 2).
- In many cases, the core tests involve a repetition of a basic set of test actions (a single symbolic test) for different test data. Therefore, we next perform system-level symbolic justification and propagation to satisfy the requirements of this symbolic test. This is very similar to symbolic RTL testability analysis for testing an RTL element (*e.g.*, functional unit, register, multiplexer, *etc.*) in a standalone core with a symbolic test vector. Hence, we adopted the *regular expression* based symbolic testability analysis scheme from [9] for this purpose. Note that the analysis scheme in [9] was applied to individual cores, not SOCs. In general, any other high-level testability analysis scheme can also be used to determine the system-level test actions for a single symbolic test. For such cases, the solution capturing the cycle-by-cycle test actions is simply equivalent to an FSA.
- Unlike the analysis which terminates at this juncture for a standalone core, we need to compose a sequence of SOC tests at time-instants dictated by the test models. If such a sequence is not realizable with the existing transparency and connectivity, we employ additional test hardware, *e.g.*, clock gating or system-level test multiplexers, to relax the core test requirements and output a low-cost solution. Finally, we employ the framework provided in [9] to minimize the test hardware added.

Composing a test sequence

We now propose a Boolean satisfiability based framework for composing a test sequence from a single symbolic test. We first illustrate our method with the help of some simple examples.

Example 5: Consider, for example, the FSA for a single symbolic test shown in Figure 10(a). If the system operates according to the sequence of actions specified by this FSA, we achieve the test objective when the system enters the accept state *SN*. Let *tN* denote the time-instant associated with state *SN* assuming the time-scale starts with state *S*0 at *t*0. Now, suppose that the test requirements specify that the test objective must be achieved every two cycles in the steady-state. In other words, we require the system to enter accept state *SN* at time-instants *tN*, $tN+2$, $tN+4$, *etc*. This, in turn, is possible if we can pipeline the FSA as shown in Figure 10(b). From the time-chart shown, it is evident that we can realize the sequence of tests if and only if states *SN*, $SN-2$, *etc.*, co-exist, and, states $SN-1$, $SN-3$, *etc.*, also co-exist. In other words, we merely need to check if the odd-numbered group of states and the even-numbered group of states are *compatible*

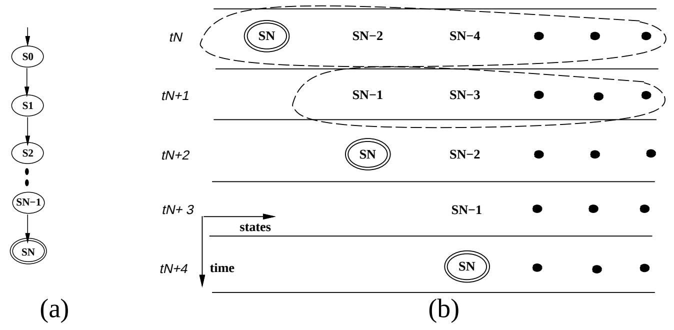


Figure 10: (a) An FSA for a symbolic test, and (b) a time-chart showing multiple tests phased in time

(a formal definition follows) for realizing the given test objective every two cycles. ∎

From the above example, we can infer that a sequence of test objectives can be met, provided that (a) individual test objectives are satisfied, and (b) some states are *compatible* with other states, with *compatibility* formally defined below.

Definition 2 : Two states S_i and S_j of an FSA A are *compatible* if and only if there exists an input transition to S_i performing a set of operations that can concurrently overlap with the operations associated with an input transition to S_j.

In the next example, we will study the additional issues that must be considered when the FSA for a single symbolic test is available. We will also motivate why a satisfiability-based approach forms a natural solution to the problem.

Example 6 : Figure 11(a) shows an FSA representing a single symbolic test that consists of a start state $S0$ and accept states $S3$, $S4$ and $S5$, with the compatibility relationships among the different states given as a compatibility graph in Figure 11(b). The compatibility graph has an edge between two states if and only if the states are compatible.

Consider the problem of scheduling successive tests at time-instants 3, 4, 5, *etc*. To accomplish this, let us instantiate the single-test FSA multiple times (say 5), as shown in Figure 11(c) (ignore the dotted encirclements for now). Clearly, we can realize the multi-test objective shown only if we can realize objectives $TEST1$, $TEST2$, $TEST3$, $TEST4$, and $TEST5$. We can rewrite this statement in the conjunctive form with Boolean variables as follows.

$$TESTS = \bigwedge^{i=1,\cdots,5} TESTi \qquad (1)$$

Since $TESTi$ is achieved if a path to the accept state from the start state exists in its FSA instantiation, we can construct an existential Boolean expression for $TESTi$ as follows. Consider $TEST1$. $TEST1$ is 1 only if state $S0$ exists at time-instant 1 (which can be represented by a Boolean variable $S0_1^1$, where the subscript to $S0$ denotes the test index and the superscript denotes the time-instant), and if one of states $S1$ or $S2$ exists at time-instant 2 (giving rise to the Boolean expression $(S1_1^2 \vee S2_1^2) \wedge (S1_1^2 \rightarrow \overline{S2_1^2})$), and so on. Additionally, only the transitions specified in the time-chart must be considered. For example, state $S1$ existing at time-instant 2 implies that state $S0$ exists at time-instant 1. This implication automatically translates to the Boolean clause $(S1_1^2 \rightarrow S0_1^1)$. The conjunction of the clauses thus obtained forms $TEST1$. Consequently, $TESTi$, for $i = 1,\cdots,5$, is obtained as follows.

$$\begin{aligned} TESTi \;=\; & (S0_i^i) \wedge (S1_i^{i+1} \vee S2_i^{i+1}) \wedge (S1_i^{i+1} \rightarrow \overline{S2_i^{i+1}}) \wedge (S3_i^{i+2} \vee \\ & S4_i^{i+2} \vee S5_i^{i+2}) \wedge (S3_i^{i+2} \rightarrow \overline{S4_i^{i+2}}) \wedge (S3_i^{i+2} \rightarrow \overline{S5_i^{i+2}}) \wedge \\ & (S4_i^{i+2} \rightarrow \overline{S5_i^{i+2}}) \wedge (S1_i^{i+1} \rightarrow S0_i^i) \wedge (S2_i^{i+1} \rightarrow S0_i^i) \wedge \\ & (S3_i^{i+2} \rightarrow S1_i^{i+1}) \wedge (S4_i^{i+2} \rightarrow S1_i^{i+1}) \wedge (S5_i^{i+2} \rightarrow S2_i^{i+1}) \end{aligned} \qquad (2)$$

```
FSA Compose_TestSeq(array<FSA> Tests,
   array<Graph> CG, FSA CoreTest, int Win){
1     TimeChart = Phase_Tests(Tests, CoreTest, Win);
2     TESTS = Clause_Gen(TimeChart, CG);
3     Soln = SAT_Solve(TESTS);
4     Sch = Schedule(Tests, CoreTest, Soln);
5     return Sch;}
```

Figure 12: Pseudocode for composing a sequence of tests

For the Boolean expression $TESTS$ in Equation (1) to be complete, we must also consider the constraints due to the compatibility graph (see Figure 11(b)). Specifically, the incompatibility of state $S0$ with state $S3$ translates to the Boolean expression $\bigwedge_{i=1}^{5}\bigwedge_{j=1}^{5}(S0_i^i \rightarrow \overline{S3_j^i}) \wedge (S3_i^{i+2} \rightarrow \overline{S0_j^{i+2}})$. The Boolean expression $INCOMPAT$ below captures the incompatibility relationships as given by the compatibility graph.

$$\begin{aligned} INCOMPAT \;=\; & \bigwedge_{i=1}^{5}\bigwedge_{j=1}^{5} (S0_i^i \rightarrow \overline{S3_j^i}) \wedge (S1_i^{i+1} \rightarrow \overline{S2_j^{i+1}}) \wedge (S1_i^{i+1} \\ & \rightarrow \overline{S3_j^{i+1}}) \wedge (S1_i^{i+1} \rightarrow \overline{S4_j^{i+1}}) \wedge (S2_i^{i+1} \rightarrow \overline{S1_j^{i+1}}) \\ & \wedge (S2_i^{i+1} \rightarrow \overline{S3_j^{i+1}}) \wedge (S2_i^{i+1} \rightarrow \overline{S5_j^{i+1}}) \wedge (S3_i^{i+2} \\ & \rightarrow \overline{S0_j^{i+2}}) \wedge (S3_i^{i+2} \rightarrow \overline{S1_j^{i+2}}) \wedge (S3_i^{i+2} \rightarrow \overline{S2_j^{i+2}}) \\ & \wedge (S3_i^{i+2} \rightarrow \overline{S4_j^{i+2}}) \wedge (S3_i^{i+2} \rightarrow \overline{S5_j^{i+2}}) \wedge (S4_i^{i+2} \rightarrow \\ & \overline{S1_j^{i+2}}) \wedge (S4_i^{i+2} \rightarrow \overline{S3_j^{i+2}}) \wedge (S4_i^{i+2} \rightarrow \overline{S5_j^{i+2}}) \\ & \wedge (S5_i^{i+2} \rightarrow \overline{S2_j^{i+2}}) \wedge (S5_i^{i+2} \rightarrow \overline{S3_j^{i+2}}) \wedge (S5_i^{i+2} \\ & \rightarrow \overline{S4_j^{i+2}}) \end{aligned} \qquad (3)$$

Finally, we can rewrite the expression for $TESTS$ as follows:

$$TESTS \;=\; (\bigwedge^{i=1,\cdots,5} TESTi) \wedge INCOMPAT \qquad (4)$$

where the expressions for $TESTi$ and $INCOMPAT$ are given in Equations (2) and (3), respectively. Solving for satisfiability of $TESTS$ determines if the sequence of test objectives can be realized. In this particular case, $TESTS$ is satisfiable (with variables $S0_1^1$, $S1_1^2$, $S4_1^3$, $S0_2^2$, $S2_2^3$, $S5_2^4$, $S0_3^3$, $S1_3^4$, $S4_3^5$, $S0_4^4$, $S2_4^5$, $S5_4^6$, $S0_5^5$, $S1_5^6$ and $S4_5^7$ being 1). This solution is pictorially represented by the dotted encirclements shown in Figure 11(c). Test schedule $Sch1$ shown in Figure 11(d) is a direct translation of the time-chart and the satisfiability solution. A state in the test schedule is a tuple of states existing at that time-instant for some instantiation in the time-chart. However, this schedule is a solution only for a finite number (five) of test objectives. We can extend this schedule to the infinite case by comparing states for equivalence. We note that state $(S4,S2,S0)$ repeats in $Sch1$. Using state equivalence, we can obtain the compacted schedule $Sch2$, as shown in Figure 11(d). Schedule $Sch2$ clearly satisfies an infinite number of test objectives. ∎

The pseudocode for SOC test sequence composition is given in Figure 12. The function *Compose_TestSeq* takes as its input an array of FSA *Tests* which must be phased according to the test requirements *CoreTest*. The compatibility graph for each FSA in *Tests* is precomputed using Definition 2, and is passed as the input array *CG*. The function *Phase_Tests* generates the time-chart *TimeChart* using the FSA *Tests* and the specific test requirements *CoreTest* (statement **1**). *Clause_Gen* (statement **2**) uses *TimeChart* and *CG* to construct the set of conjunctive clauses *TESTS*, as described in Example 6. *Sat_Solve* (statement **3**) next checks *TESTS* for satisfiability and if satisfiable, the function *Schedule* (statement **4**) returns a valid test schedule.

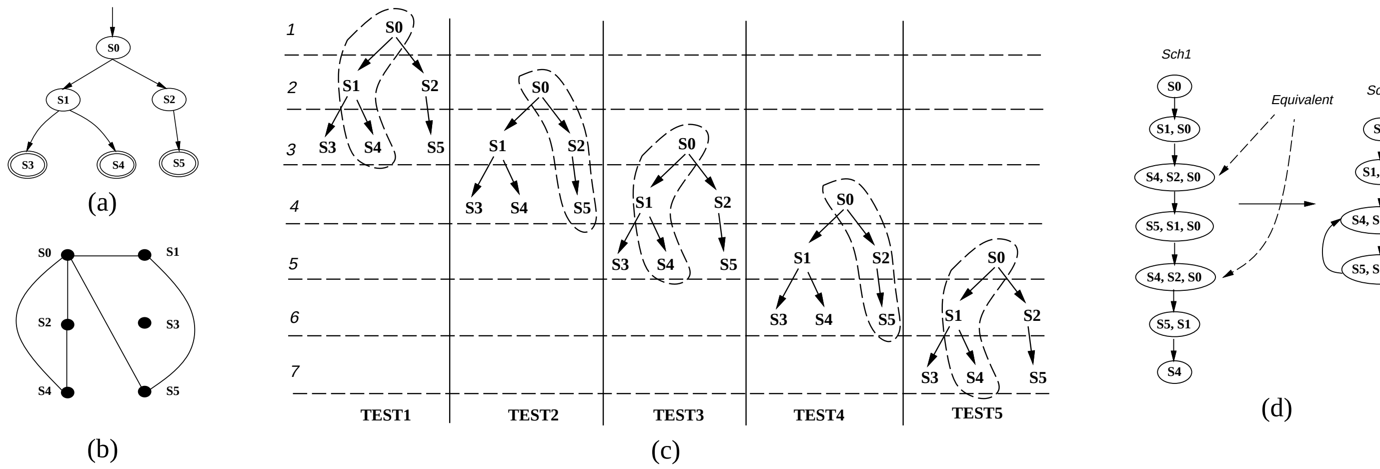


Figure 11: (a) A single symbolic test, (b) its compatibility graph, (c) time-chart for multiple-test objectives, and (d) final test schedules

Table 1: Testability results

SOC	*Area*			*Tapp*		
	Orig	*Mod*	*Ovhd* (%)	*[6]* (cyc)	*Our* (cyc)	*Red* (%)
Sys_Gen	666330	678807	1.87	4620	2460	46.75
Grid	1225592	1330840	8.59	34312	7184	79.06
*Star*4	730881	773175	5.79	68224	7184	89.47
Mesh	956044	967800	1.23	46629	8197	82.42
*Star*8	918312	957620	4.28	271696	13968	94.86

The solution scheme outlined in Example 6 phases a single symbolic test infinite number of times. For this example, a small window of tests is sufficient to determine a valid test schedule (existence of one or otherwise). Since the actual bound on window size is theoretically exponential in the number of states, our algorithm starts with an initial window size value and explores the solution space until a user-specified limit, *Win*, is reached. If a null test schedule is returned in the process, test hardware, such as clock gating or test multiplexer, is added to relax the core test requirements, and symbolic test generation is repeated.

4 Experimental Results

We next present experimental results obtained by applying our algorithm to some example SOCs. The SOC *Sys_Gen* was seen in Section 2. SOCs *Grid*, *Star*4, *Mesh* and *Star*8 are systolic architectures proposed to study the performance of digital signal processing applications. They consist of processor cores connected in different configurations to effect pipelined processing of input data.

The testability results are given in Table 1. Columns 2 and 3 give the area of the SOCs before and after running our algorithm. The area numbers are actual layout numbers generated after placement and routing with the Octtools package from University of California, Berkeley. Column 4 reports the resultant area overheads for these SOCs, with an average of only 4.4%. Columns 5 and 6 compare the test application time for our approach with the one in [6], which drastically reduces test application time compared to the traditional approaches. Column 7 gives the percentage reduction in the test application time, with an average of 78.5%. Since the scheme in [6] cannot handle some of the SOCs, we conservatively extended their approach to estimate the test application time. Our testability approach achieves 100% test coverage of all embedded cores in all SOCs. For example, the test architecture derived for the SOC *Sys_Gen* (see Figure 4) provides complete access to apply the scan tests for the core TLU as well as the precomputed test sequences for cores CU and $DG1$. In this way, all embedded cores are completely tested with the precomputed test sets (or sequences) provided for them.

5 Conclusions

We provided a comprehensive framework for analyzing the testability of core-based SOCs for generating low-overhead test architectures and compact test schedules. Salient features of this work include the modeling of transparency, tests and test hardware using finite-state automata, and providing a rigorous system-level testability analysis framework. Experimental results show complete test coverage with low area overheads and test application times.

References

[1] P. Varma, "System chip test: Are we there yet?," in *Proc. Int. Test Conf.*, p. 1144, Oct. 1998.

[2] Y. Zorian, E. J. Marinissen, and S. Dey, "Testing embedded-core based system chips," in *Proc. Int. Test Conf.*, pp. 130–143, Oct. 1998.

[3] N. Touba and B. Pouya, "Using partial isolation rings to test core-based designs," *IEEE Design & Test of Computers*, vol. 14, no. 4, pp. 52–59, 1997.

[4] P. Varma and S. Bhatia, "A structured test re-use methodology for systems on silicon," in *Proc. IEEE Int. Wkshp. Testing Embedded Core-Based Systems*, Nov. 1997.

[5] F. Bouwman, S. Oostdijk, R. Stans, B. Bennetts, and F. Beenker, "Macro Testability: The results of production device applications," in *Proc. Int. Test Conf.*, pp. 232–241, Nov. 1992.

[6] I. Ghosh, N. K. Jha, and S. Dey, "A low overhead design for testability and test generation technique for core-based systems," in *Proc. Int. Test Conf.*, pp. 50–59, Nov. 1997.

[7] I. Ghosh, S. Dey, and N. K. Jha, "A fast and low cost testing technique for core-based system-on-chip," in *Proc. Design Automation Conf.*, pp. 542–547, June 1998.

[8] M. Sipser, *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[9] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression based high-level testability analysis and optimization," in *Proc. Int. Test Conf.*, pp. 331–340, Oct. 1998.

[10] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proc. European Design Automation Conf.*, pp. 214–218, Feb. 1991.