Fast Enhancement of Validation Test Sets to Improve Stuck-at Fault Coverage for RTL circuits

L. Lingappan†, V. Gangaram‡, N. K. Jha† and S. Chakravarty††
†Princeton University, Princeton, NJ 08544
{llingapp, jha}@princeton.edu
‡ Intel Corporation, Folsom, CA 95630
vijay.gangaram@intel.com
†† Cswitch Inc., Santa Clara, CA 95054
sreejit@ieee.org

***Abstract*— A digital circuit usually comprises a controller and datapath. The time spent for determining a valid controller behavior to detect a fault usually dominates test generation time. A validation test set is used to verify controller behavior and, hence, it activates various controller behaviors. In this paper, we present a novel methodology wherein the controller behaviors exercised by test sequences in a validation test set are reused for detecting faults in the datapath. A heuristic is used to identify controller behaviors that can justify/propagate pre-computed test vectors/responses of datapath register-transfer level (RTL) modules. Such controller behaviors are said to be *compatible* with the corresponding pre-computed test vectors/responses. The heuristic is fairly accurate, resulting in the detection of a majority of stuck-at faults in the datapath RTL modules. Also, since test generation is performed at the RTL and the controller behavior is predetermined, test generation time is reduced. For microprocessors, if the validation test set consists of instruction sequences then the proposed methodology also generates instruction-level test sequences.**

I. Introduction

Validation/functional testing methods [1]–[7] start with a functional description of the circuit and make sure that the circuit's operation corresponds to its description. These methods are geared towards extensive validation of the controller behavior of a given circuit under test. However, validation test sets do not have good stuck-at fault coverage. In this paper, we present a novel method that uses validation test sets to generate test sequences that detect a majority of stuck-at faults in the datapath.

A. Related Work

A digital system, in general, consists of two main parts: a datapath and controller. The datapath is used to store and manipulate data and transfer data from one part of the system to another. The controller, on the other hand, controls the operation of the datapath. In practice, obtaining a suitable controller behavior to detect a fault is much more difficult than performing justification/propagation analysis on the datapath alone [8] and, hence, constitutes a predominant portion of the test generation time. Various schemes [9], [10] have been suggested to avoid controller analysis during test generation by inserting DFT elements at the interface between the controller and datapath. DFT elements can also be inserted in the controller to simplify its functionality in the test mode [11].

Design validation deals with verifying the conformance of the design to its functional specification, thereby aiding in the elimination of design errors. A functional coverage metric is often used that measures the fraction of specified behaviors exercised by a test suite. A majority of the design validation schemes [2]–[4] use the percentage of state transition edges in the controller that are exercised by a test suite as the functional coverage metric. Hence, all the above validation schemes generate test sets that exercise the controller in different ways to detect design errors.

Acknowledgments: This work was supported by SRC under Contract No. 2002-TJ-1039.

B. Paper Overview and Contributions

In this paper, we define the problem of identifying pre-computed test vectors/responses of datapath RTL modules that can be justified/propagated to primary inputs/outputs by reusing the controller behavior derived from a validation test sequence. Such a controller behavior is defined to be *compatible* with the corresponding pre-computed test vectors/responses. The advantages of the proposed approach are as follows.

- Hardware design validation, in spite of being an NP-complete problem, is an important step in the early stages of the integrated circuit design flow. The proposed approach exploits the availability of a validation test set for high-level testability analysis.
- The controller behavior is pre-determined during test generation. Hence, unlike conventional sequential test generation approaches, the number of time frames for which a sequential circuit is unrolled is fixed.
- Test generation is done at the RTL. Also, since the controller behavior is pre-determined, test generation time is reduced.
- The controller behavior is preserved during test generation for justifying/propagating pre-computed test vectors/reponses of datapath modules. Hence, for microprocessors, if the validation test set consists of instruction sequences, then the proposed approach also generates a test program consisting of instruction sequences.

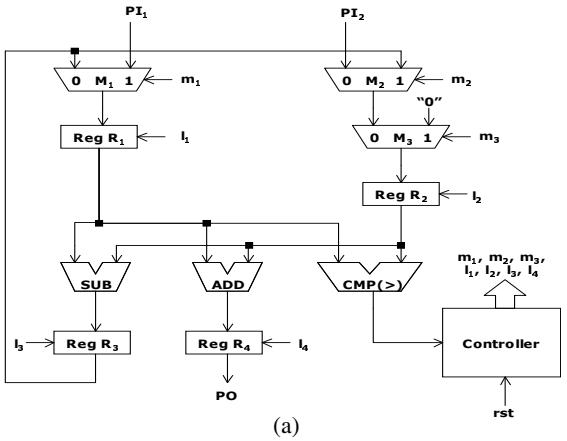
Experimental results for a wide range of benchmark circuits show that the proposed approach is efficient in generating tests that reuse controller behaviors extracted from validation test sequences and obtain high fault coverages for datapath RTL modules.

The rest of the paper is organized as follows. Section II motivates the proposed approach by illustrating its advantages. Section III defines and illustrates the heuristic for quickly identifying controller behaviors that can be reused for justifying/propagating pre-computed test vectors/responses. Section IV presents experimental results, while Section V concludes.

II. Motivation

In this section, we illustrate the proposed approach and motivate the need for a heuristic that efficiently identifies controller behaviors extracted from validation test sequences that can be reused successfully for justifying/propagating pre-computed test vectors/responses of an embedded RTL module to primary inputs/outputs of the circuit containing the RTL module.

Consider the RTL circuit *ascmp* shown in Fig. 1(a). The circuit comprises registers R_1, R_2, R_3 and, R_4, primary inputs PI_1, PI_2 and reset *rst*, constant input "0", primary output *PO*, multiplexers M_1, M_2 and M_3, comparator ($>$) *CMP*, adder *ADD*, subtracter *SUB* and a controller. The bit-width of all



(a)

rst	cmp	PS	NS	m_1	m_2	m_3	l_1	l_2	l_3	l_4
0/1	X	S0	S1/S0	1	0	1	1	1	0	0
0/1	0	S1	S0	0	0	0	0	0	0	1
0/1	1	S1	S2/S0	0	0	0	0	1	0	0
0/1	0	S2	S0	0	0	0	0	0	0	1
0/1	1	S2	S3/S0	0	0	0	0	0	1	0
0/1	X	S3	S4/S0	0	0	0	1	1	0	0
0/1	0	S4	S0	0	0	0	0	0	0	1
0/1	1	S4	S5/S0	0	0	0	0	0	1	0
0/1	X	S5	S6/S0	1	1	0	1	1	0	0
0/1	X	S6	S0	0	0	0	0	0	0	1

(b)

Fig. 1. (a) The RTL circuit *ascmp*, and (b) controller specification for *ascmp*

modules in the datapath is four. The controller, shown in Fig. 1(b), consists of seven states, $S0$, $S1$, $S2$, $S3$, $S4$, $S5$, and $S6$, and generates the control signals shown in Fig. 1(a). The next state (NS) and values at the outputs of the controller are determined by its present state (PS), *CMP*'s output and *rst*. Note that if *rst* is asserted then NS is always $S0$ irrespective of PS and the value at *CMP*'s output.

Assume that T_1 is a validation test sequence consisting of the following six vectors: {(100000000), (000010000), (000000000), (000000000), (000000101), (000000000)}. Each vector consists of nine bit values held by different primary inputs in the following order: rst, $PI_1[3:0]$, $PI_2[3:0]$. The behavior of the controller is governed by the values held by the input signals to the controller (*rst* and *CMP*'s output). These values are obtained by performing logic simulation on the circuit *ascmp* using sequence T_1. In order to reuse the controller behavior, the following conditions are imposed during test generation: (i) the circuit is unrolled for the same number of cycles as the number of vectors in the validation test sequence, and (ii) the values held by the controller inputs in each cycle are determined by logic-simulating the circuit using the validation test sequence.

Fig. 2 shows the control-data flow graph (CDFG) and state transition sequence exercised by validation test sequence T_1. The CDFG captures the data flow through different modules of the datapath. The Boolean values shown at the output of comparator module *CMP* are obtained through logic simulation of test sequence T_1. If the controller behavior derived from T_1 is reused to generate a new test sequence T_2 that targets stuck-at faults, then the state transition sequence and CDFG corresponding to T_2 will be identical to those for T_1 shown in Fig. 2.

Next, assume that the RTL module *ADD* in Fig. 1(a) is the module under test. RTL test generation is SAT based [12] and

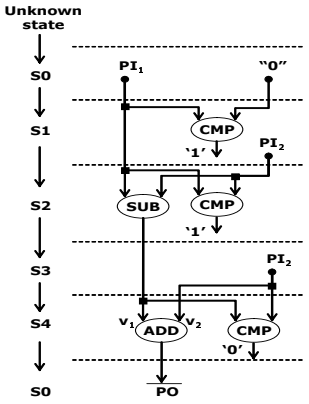


Fig. 2. CDFG and state transition sequence exercised by validation test sequence T_1

uses a pre-computed test set approach [10], [13]. Justification of module-level pre-computed test vectors is performed by (i) first abstracting modules by their equivalent input/output propagation rules, (ii) abstracting the different paths in the circuit as a sequence of implications, and (iii) using a SAT solver to resolve the implications to obtain requisite vectors at the primary inputs.

A pre-computed test vector for *ADD* consists of two sub-vectors (v_1, v_2) corresponding to the two inputs of the module. *ADD* uses a carry-lookahead implementation for computing the sum of the values held by its two inputs. A pre-computed test set that detects all the stuck-at faults in *ADD* consists of the following vectors: {(15,1), (1,15), (2,14), (5,12), (8,9), (0,15), (14,0), (13,1), (11,3), (7,7)}. The goal of RTL test generation is to justify these vectors from the inputs of *ADD* to primary inputs PI_1, PI_2, and *rst*, and propagate the responses to primary output *PO*. The controller behavior derived from test sequence T_1 is reused during test generation by (i) ensuring that the sequential circuit is unrolled six times (T_1 consists of six vectors) and (ii) maintaining the same values for signals *rst* and *CMP* as imposed by test sequence T_1.

In Fig. 2, a pre-computed test vector (v_1, v_2) can be delivered to *ADD* only in state $S4$ and the response can be observed only in the subsequent state $S0$. However, in general, a given module can be exercised in multiple states by a given validation test sequence. For example, *CMP* in Fig. 2 is exercised in states $S1$, $S2$ and $S4$. By reusing the controller behavior extracted from T_1, five out of ten vectors in the pre-computed test set ({(1,15), (2,14), (5,12), (8,9), (7,7)}) can be delivered to the inputs of *ADD* and their responses observed at the primary output. The controller behavior extracted from T_1 is not compatible for test generation using the remaining pre-computed vectors {(15,1), (0,15), (14,0), (13,1), (11,3)}. While vectors {(15,1), (14,0), (13,1), (11,3)} cannot be justified due to the Boolean value held by the output of *CMP* in state $S4$ in Fig. 2, pre-computed vector (0,15) cannot be justified due to the requirement that the value at the output of *CMP* in state $S2$ should be '1'. Hence, reusing controller behavior extracted from T_1 results in successful test generation for 50% of the pre-computed test vectors of *ADD*. In general, a validation test set may contain a large number of test sequences. For example, assume that the validation test set of *ascmp* consists of 20 sequences. The total number of pre-computed test vectors for all the datapath RTL modules in *ascmp* is 58. Hence, in the worst case, 1160 (58×20) test generation runs have to be attempted to determine whether there exists a controller behavior that can be reused to justify/propagate each pre-computed test vector/response. This problem worsens if the average length of the validation test

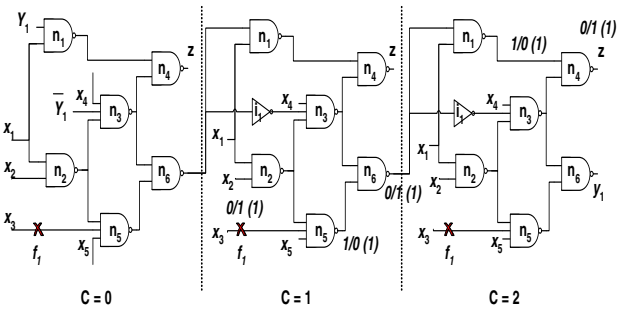


Fig. 3. Activation and detection cycles for fault f_1 obtained by fault simulation

sequences is high. Even though reusing controller behavior and test generation at the RTL reduce the time required for each test generation run, performing test generation for each pre-computed test vector by reusing the controller behavior extracted from each validation test sequence till all pre-computed test vectors/responses get justified/propagated might result in a prohibitively large number of test generation runs. In the next section, we present a heuristic that identifies efficiently whether the controller behavior extracted from a validation test sequence is compatible for justifying/propagating a pre-computed test vector/response to primary inputs/outputs.

III. Identification of Compatible Controller Behaviors

In the previous section, we showed that a pre-computed test vector/response of an embedded RTL module in the datapath of a circuit can be justified/propagated by reusing the controller behavior extracted from a validation test sequence. Such a controller behavior is defined to be compatible with the pre-computed test vector/response. Next, we present a heuristic to quickly identify a compatible controller behavior for a given pre-computed test vector/response. Hence, test generation is performed only for those pre-computed test vectors/responses for which a compatible controller behavior has been selected by the heuristic. This significantly reduces the number of test generation attempts required to justify/propagate the pre-computed test vectors/responses of all the RTL modules in the datapath of a circuit.

The heuristic consists of two steps. The validation test sequences are first fault-simulated. The stuck-at fault simulator is augmented to capture the cycles in which a fault gets activated and eventually detected. If a detected fault belongs to an RTL module in the datapath, then the second part of the heuristic is used to estimate whether the controller behavior extracted from the corresponding validation test sequence is compatible with any one of the module's pre-computed test vectors/responses. This is done under the assumption that pre-computed test vectors (responses) will take the same number of cycles to get justified (propagated) as taken for the detected stuck-at fault. We next present the two steps of the heuristic in detail.

A. Augmented Fault Simulation to Derive Activation-Detection Time Frame Pair

We modified the HOPE fault simulator [14] in order to obtain the activation-detection cycles as a by-product of fault simulation. HOPE is a hybrid of concurrent, differential and parallel fault simulation algorithms. It is based on an iterative-logic array (ILA) model of the sequential circuit. Upon activation, a fault gives rise to a fault effect which is propagated through the ILA model of the sequential circuit one cycle at a time until it either reaches a primary output or is blocked due to circuit conditions or length of the test sequence.

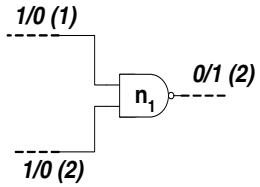


Fig. 4. Special case for determining activation and detection cycles

With each fault effect, we associate a tag which is an integer data structure that stores the activation cycle of the fault effect. The tag gets propagated through the ILA along with the fault effect. When the fault effect reaches a primary output port, the fault simulator declares the fault detected, reports the activation cycle stored in the tag and current cycle of the ILA as the detection cycle. Once the fault is detected, it is dropped from the fault list.

Example 1: Fig. 3 shows the ILA model of a sequential circuit over three cycles, $C = (0, 1, 2)$. The circuit comprises six NAND gates n_1, n_2, n_3, n_4, n_5 and n_6, five primary inputs x_1, x_2, x_3, x_4, and x_5, a primary output z and a flip-flop. Y_1 and y_1 represent the input and output of the flip-flop, respectively. Assume a stuck-at 1 (SA1) fault at primary input x_3 and a test sequence $\{(00110), (10001), (11101)\}$ that detects the fault. Each test vector corresponds to the values held by primary inputs $(x_1, x_2, x_3, x_4, x_5)$ in that order. The fault is present in all the three cycles and gets activated in cycle $C = 1$, giving rise to the fault effect $0/1$ (good-circuit value/faulty-circuit value). The activation cycle, $C = 1$, is stored in an integer tag (shown within brackets in Fig. 3) along with the fault effect. The tag gets propagated along with the fault effect to primary output z in cycle $C = 2$. At this point, the fault simulator checks the tag associated with the fault effect and declares that fault f_1 was activated in cycle $C = 1$ and detected in cycle $C = 2$. ∎

The above example does not capture the special case shown in Fig. 4. Here, a fault gets activated in two different cycles, $C = 1$ and $C = 2$. These fault effects get propagated to a cycle $C \geq 2$ and meet at a NAND gate n_1. Both the fault effects get propagated to the output of gate n_1. In this case, the tag associated with the fault effect at the output of gate n_1 is assigned the greater of the two cycle values, i.e., $C = 2$. This is because, in general, activating a fault is easier than propagating the corresponding fault effect in a circuit and choosing the larger activation cycle reduces the number of cycles for which propagation analysis needs to be done.

In the next section, we present a heuristic that uses the information derived from the augmented fault simulator to identify compatible controller behaviors that can be reused to successfully justify/propagate pre-computed test vectors/responses of an embedded RTL module.

B. Analysis of Requirements to Identify Compatible Faults

Given a validation test sequence, we first extract the CDFG exercised by the sequence. Fig. 2 shows one such CDFG extracted for circuit *ascmp*, shown in Fig. 1, using validation test sequence T_1. We next need to determine efficiently and accurately whether justification/propagation of any pre-computed test vector/response of an RTL module in the CDFG will result in a failure. For this purpose, we define two measures called *1-requirement* (r_1) and *0-requirement* (r_0) for each line in the CDFG. They capture approximate requirements for a line to hold binary value '1' or '0', respectively, for justifying/propagating a pre-computed test vector/response in the CDFG. The range of values that r_0 and r_1 can assume is $0 \leq r_0,\ r_1 \leq 1$. Also, for any line, if $r_1(r_0) > 0$ then $r_0(r_1) = 0$. Given a set of initial assignments, the non-zero requirement values are justified to primary inputs in the CDFG. The initial

0-7695-2762-0/07 $20.00

```
JUSTIFICATION PROCESS (CDFG, initial (r0, r1) values)
 1 : Levelize circuit; L ← max_level
 2 : Q[L]: Array of queues //Each queue stores the initial
     assignment of non-zero requirement values per level
 3 : for i = 0 to L
 4 :     while (Q[i] non_empty)
 5 :         {o, r0(o), r1(o)} = pop Q[L]
 6 :         Justify r0(o), r1(o)
 7 :         for each input i of gate G //G is the gate whose
                                        output is line o
 8 :             if (r0(i) > 0) AND (r1(i) > 0)
 9 :                 evaluate γ
10:                  if γ > γ0 //γ0 = 1.0
11:                      Declare incompatible; STOP
12:                  else
13:                      if r1(i) > r0(i)
14:                          r0(i) = 0
15:                      else
16:                          r1(i) = 0
17:              push {i, r0(i), r1(i)} to Q[level(i)]
18: Declare compatible
```

Fig. 5. Algorithm for justifying requirement values

assignments correspond to a pre-computed test vector/response and the values at the controller inputs. Justification through RTL modules is performed using their gate-level implementation. The justification process obeys the following rules for gate-level modules:

- AND gate: Let the number of gate inputs be n. Let i and o represent the gate input and output lines, respectively. Then
  $if\ r_0(i) < r_0(o)/n,\ then\ r_0(i) = r_0(o)/n$
  $if\ r_1(i) < r_1(o),\ then\ r_1(i) = r_1(o)$
- OR gate: Let the number of gate inputs be n. Let i and o represent the gate input and output lines, respectively. Then
  $if\ r_0(i) < r_0(o),\ then\ r_0(i) = r_0(o)$
  $if\ r_1(i) < r_1(o)/n,\ then\ r_1(i) = r_1(o)/n$
- NOT gate: Let i and o represent the gate input and output lines, respectively. Then
  $if\ r_0(i) < r_1(o),\ then\ r_0(i) = r_1(o)$
  $if\ r_1(i) < r_0(o),\ then\ r_1(i) = r_0(o)$
- Fan-out point: Let a line i have k fan-outs $o_1, o_2, \ldots, o_k$. Then
  $r_0(i) = max(r_0(o_p),\ r_0(i))$
  $r_1(i) = max(r_1(o_p),\ r_1(i)),\ 1 \le p \le k$

The CDFG is levelized before the justification process. Hence, a single pass is sufficient to justify the requirement values to the primary inputs. In some cases, during the justification process a condition might arise wherein the 0- and 1-requirements of a particular line are both assigned values greater than 0. Such a conflicting situation arises when a given line in a CDFG might need to hold both Boolean values '0' and '1' in order to satisfy the initial assignment of requirement values. The extent of the conflict on a line l is measured with the help of the following metric γ.

$$\gamma = \frac{1}{2^{1-(r_0(l)*r_1(l))} - 1} - 1$$

When either $r_0(l)$ or $r_1(l)$ hold the value 0, γ evaluates to 0, indicating no conflict. However, when both $r_0(l)$ and $r_1(l)$ are equal to 1, γ evaluates to infinity indicating that l needs to hold both complementary Boolean values, which is not possible. Also, γ is a monotonically increasing function in both $r_0(l)$ and $r_1(l)$ in the range $0 < r_0(l),\ r_1(l) \le 1$. Based on the experiments presented in Section IV, a threshold value of 1.0 was selected for γ. Hence, if γ for a conflicting situation is greater than 1.0, then justification is stopped and the corresponding controller behavior is declared to be incompatible with the pre-computed test vector/response. However, if $\gamma \le 1.0$, then the smaller of the two requirement values is changed to 0.

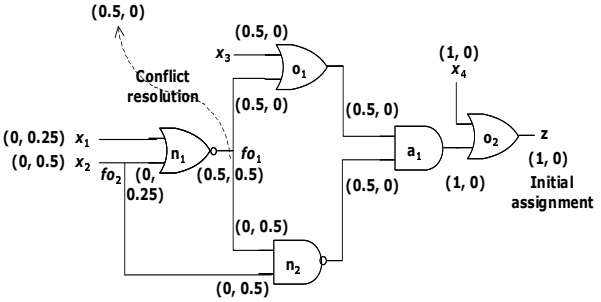


Fig. 6. Justification of requirement values

The justification process is summarized by the pseudocode shown in Fig. 5. Q is an array of queues that stores requirement values of lines that need to be justified. The values are stored based on the levels of the corresponding lines. The values are justified starting from lines with the lowest level number (primary outputs in the CDFG) to lines that have the maximum level number (primary inputs in the CDFG). Step 6 in Fig. 5 is performed using the rules described above for gate-level modules. In case of a conflict on a line o (both $r_0(o),\ r_1(o) > 0$), γ is evaluated. Depending on its value, we either continue with the justification process or declare the controller behavior corresponding to the CDFG as incompatible with the targeted pre-computed test vector/response. If the justification process completes with all intermediate γ values, if any, falling below the threshold value (γ_0), then the controller behavior is declared to be compatible with the pre-computed test vector/response. In this case, we proceed to the test generation phase. Next, we present an example that further illustrates the justification process.

Example 2: Consider the gate-level circuit shown in Fig. 6. It consists of one NOR gate n_1, one NAND gate n_2, one AND gate a_1, two OR gates o_1 and o_2, four primary inputs $x_1,\ x_2,\ x_3$ and x_4, one primary output z, and two fan-out nodes fo_1 and fo_2. The requirement values are captured in the format (r_0, r_1). Assume that an initial requirement value assignment of (1,0) is made at z, while the requirement values for all the other nodes are initialized to (0,0). The initial non-zero assignment at z is justified using the procedure shown in Fig. 5 to the primary inputs. In other words, we want to quickly estimate whether there exists an assignment of Boolean values to primary inputs that imply a Boolean value '0' at z. Fig. 6 shows the requirement values at different nodes at the end of the justification process. A conflict arises at fan-out node fo_1, since both $r_0(fo_1)$ and $r_1(fo_1)$ get assigned a value of 0.5. γ for this conflict evaluates to 0.46. Since γ is less than γ_0 (1.0), the conflict is resolved using steps 13-16 in Fig. 5. In this case, the heuristic predicts that there exist primary input value assignments that resolve this conflict. This is indeed true as there are three vectors $(x_1, x_2, x_3, x_4) = \{(0100), (1000), (1100)\}$ that result in a Boolean value '0' at z. The requirement values at all the nodes can be verified by using the justification rules for gate-level modules described above. ∎

The above heuristic is used to estimate whether the controller behavior shown in Fig. 2 that is derived from validation test sequence T_1 is compatible with any pre-computed test vector/response of the RTL module *ADD* in Fig. 1. We first use the augmented fault simulator described in Section III-A

to check whether T_1 detects any faults in *ADD*. It turns out that T_1 detects 12 faults in *ADD*. The activation and detection cycles indicate that these faults are activated in state *S*4 shown in Fig. 2 and detected in subsequent state *S*0. We next assume that any pre-computed test vector of *ADD* can be delivered to it only in state *S*4 and its response observed in one of the subsequent states. This assumption holds for *ADD* in Fig. 2 as it is activated only in state *S*4. However, in general, a module can get activated multiple times in a CDFG. Hence, the augmented fault simulator is used to obtain the number of cycles required to deliver a pre-computed test vector from the primary inputs to the inputs of the module under test and observe its response at a primary output. The initial non-zero requirement values correspond to the pre-computed vectors at the inputs of *ADD* in state *S*4 and the output of *CMP* in states *S*1, *S*2 and *S*4 to preserve the controller behavior. For example, for the pre-computed test vector (1,15) the 0-requirement values for the two input nodes of *ADD* are (1110,0000). The requirement values are listed in the bit order (3,2,1,0). Similarly the 1-requirement values for the two input nodes are {(0001),(1111)}. The requirement values for other pre-computed test vectors can be obtained in a similar fashion. The requirement values (r_0, r_1) for the output of *CMP* in states *S*1, *S*2 and *S*4 are (0,1), (0,1) and (1,0), respectively. For each pre-computed test vector, the initial non-zero requirement value assignments are justified using the procedure shown in Fig. 5. Only a single pass through the CDFG is required for each pre-computed test vector/response to determine whether the controller behavior corresponding to the CDFG is compatible with the pre-computed test vector/response. For the pre-computed test vectors {(1,15), (2,14), (5,12), (8,9), (7,7)}, none of the conflicts generated in the justification process result in γ being greater than 1.0. Hence, the controller behavior is declared to be compatible with each of these pre-computed test vectors. RTL test generation is then performed to justify all the above five pre-computed test vectors. For the remaining pre-computed test vectors {(15,1), (0,15), (14,0), (13,1), (11,3)}, conflicts result in the following γ values, (3.96, 4.69, 3.96, 3.96, 3.96), respectively. All these values are greater then the threshold limit (γ_0) of 1.0. Hence, the controller behavior is declared to be incompatible with all the five pre-computed test vectors and no test generation is performed for these vectors. The γ values for the pre-computed test vectors {(15,1), (14,0), (13,1), (11,3)} are equal because they cause identical conflicts at the most significant bit of the input to *ADD* from R_1 in Fig. 1(a). For *ADD*, all the pre-computed test vectors that are found to be compatible with the controller behavior also get successfully justified. None of the pre-computed test vectors that are found to be incompatible can be justified by reusing the controller behavior extracted from T_1. Hence, in this case, the heuristic is 100% accurate in efficiently determining whether a controller behavior is compatible with any pre-computed test vector/response of an embedded RTL module.

The output of *ADD* in Fig. 2 is directly connected to the primary output *PO*. Hence, no propagation analysis is required. However, in general, an RTL module's output might be connected to other modules in the CDFG. In this case, we assign additional requirement values initially to ensure that a controller behavior is compatible for both justification and propagation of pre-computed test vectors and responses, respectively. Once a controller behavior is found to be compatible with a pre-computed test vector and response, SAT based RTL ATPG [12] is used to obtain a test sequence that reuses the controller behavior to justify and propagate the pre-computed test vector and response to primary inputs and outputs, respectively. In the next section, we present experimental results for various benchmark circuits using the proposed approach.

TABLE I
CIRCUIT STATISTICS

Circuit	BW	♯lits	♯regs	♯FFs	♯gates
Sqr_mul	32	21,689	6	210	11,248
Sqr_add	32	13,563	7	226	7,623
ex1	1-16	3,907	8	130	2,017
Bessel	32	23,007	5	162	13,123
Paulin	32	76,123	5	196	39,558
ASPP4	32	101,278	7	229	52,742
Parwan	8-12	2,451	7	89	1,323

IV. EXPERIMENTAL RESULTS

In this section, we provide experimental results obtained by applying our methodology to various benchmarks. The fault simulator used in our experiments was obtained by modifying the HOPE fault simulator [14], as illustrated in Section III-A. The heuristic, explained in Section III-B, first uses a Perl script to generate CDFGs corresponding to different validation test sequences of a circuit [15]. A C++ code is then used to levelize the circuit and determine whether the controller behavior corresponding to a CDFG is compatible with a pre-computed test vector/response. The controller behaviors that are found to be compatible are reused during SAT based RTL test generation [12]. All experiments were performed on a 3.06 GHz Linux server with 2 GB of memory. Synopsys Design Compiler [16] is used to obtain the synthesized gate-level netlists.

Table I shows the circuit statistics for the benchmarks used in our experiments. Column 2 gives the bit-width of the datapath. Columns 3, 4, 5 and 6 report the literal-count of the synthesized circuit, number of registers, number of flip-flops, and number of gates, respectively. Of the seven circuits, RTL circuits *Sqr_Mul* and *Sqr_add* perform a polynomial computation of their inputs [12]. In *ex1*, the datapath input values influence the controller behavior heavily. Hence, the controller and datapath are tightly coupled in *ex1* [12]. *Bessel* is a circuit used in the computation of an integer-order Bessel function [17]. *Paulin* is a differential equation solver and is data-dominated. *ASPP4* is an application-specific programmable processor (ASPP) taken from [15] that can emulate the behavior of *Paulin*. *Parwan* [18] is a simple accumulator-based microprocessor. It has an 8-bit data bus and a 12-bit address bus for external accesses.

Table II presents results obtained using the proposed methodology for the above benchmarks. Columns 2 and 3 present the number of validation test sequences used for each benchmark and the overall fault coverage obtained by these sequences for the datapath RTL modules, respectively. These sequences were hand-derived to exercise all the state transition edges in the controller. The total number of pre-computed test vectors for the datapath RTL modules in these circuits is presented in Column 4. Column 5 indicates the number of times a controller behavior extracted from any validation test sequence is found to be compatible with a pre-computed test vector/response. Column 6 presents the number of RTL test generation runs per benchmark that are successful. Column 7 gives the total time taken to find compatible controller behaviors and perform RTL test generation runs. Column 8 presents the fault coverage obtained by the generated test sequences for the datapath RTL modules, while Column 9 presents the overall controller/datapath fault coverage.

A majority of stuck-at faults in the datapath get detected by reusing the controller behaviors extracted from the validation test sequences (Column 8). The overall fault coverage shown in Column 9 is lower than the fault coverage for the datapath RTL modules as the faults in the gate-level controller are not

TABLE II
TEST GENERATION RESULTS

Module	# validation test sequences	Initial datapath fault cov. (%)	# pre-computed test vectors	# compatible	# success justified/ propagated	Total time (in secs)	Datapath fault cov. (%)	Overall fault cov. (%)
Sqr_mul	12	40.7	62	79	62	217.8	100.0	99.2
Sqr_add	16	23.6	64	75	64	156.2	100.0	98.1
ex1	35	58.1	103	156	99	118.3	99.8	96.6
Bessel	27	47.3	79	105	76	179.9	99.7	97.2
Paulin	49	20.1	123	193	119	285.4	98.2	96.8
ASPP4	44	27.8	155	238	145	364.1	95.3	94.4
Parwan	26	56.6	123	170	97	380.3	96.9	87.4

targeted for test generation. The number of test generation runs (Column 5) is comparable to the number of pre-computed test vectors (Column 4) and a large number of these runs are also successful (Column 6). This directly reflects the accuracy of the heuristic presented in Section III which identifies controller behaviors that can be reused for justifying/propagating a given pre-computed test vector/response. The accuracy of the heuristic reduces the number of test generation runs per pre-computed test vector/response and, hence, the overall number of test generation runs for each benchmark. Also, SAT based RTL test generation reduces the overall test generation time (Column 7). For *Parwan*, the validation test set consists of instruction sequences. Since test generation preserves controller behaviors extracted from validation test sequences, the generated sequences constitute a test program consisting of 145 instructions.

V. CONCLUSIONS

In this paper, we presented a novel approach for using a validation test set to generate test sequences that have good stuck-at fault coverage for datapath RTL modules. The scheme first derives the controller behaviors from validation test sequences and reuses them for simplifying justification/propagation analysis corresponding to pre-computed test vectors/responses of datapath RTL modules. A heuristic is used to identify controller behaviors that are compatible with a given set of pre-computed test vectors/responses. It requires only a single pass through the CDFG corresponding to a validation test sequence and is accurate, resulting in a small number of test generation runs. Test generation is performed at the RTL and the controller behavior is pre-specified, which results in very small test generation times.

REFERENCES

[1] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, Cambridge, 2003.

[2] D. J. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction techniques for validation coverage analysis and test generation," *IEEE Trans. Comput.*, vol. 47, pp. 2–14, Jan. 1998.

[3] K. T. Cheng and J. Y. Jou, "A functional fault model for sequential machines," *IEEE Trans. Computer-Aided Design*, vol. 2, pp. 1065–1073, Sept. 1992.

[4] R. C. Ho and M. A. Horowitz, "Validation coverage analysis for complex digital designs," in *Proc. Int. Conf. Computer-Aided Design*, pp. 146–151, Nov. 1996.

[5] F. Fallah, P. Ashar, and S. Devadas, "Simulation vector generation from HDL descriptions for observability-enhanced statement coverage," in *Proc. Design Automation Conf.*, pp. 666–671, June 1999.

[6] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, "A study in coverage-driven test generation," in *Proc. Design Automation Conf.*, pp. 970–975, June 1999.

[7] N. Yogi and V. D. Agrawal, "Spectral characterization of functional vectors for gate-level fault coverage tests," in *Proc. VLSI Design & Test Symp.*, pp. 407–417, Aug. 2006.

[8] T. E. Marchok, A. El-Maleh, W. Maly, and J. Rajski, "Complexity of sequential ATPG," in *Proc. European Design and Test Conf.*, pp. 252–261, Mar. 1995.

[9] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design for testability technique of RTL circuits using control/data flow extraction," in *Proc. Int. Conf. Computer-Aided Design*, pp. 329–336, Nov. 1996.

[10] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression-based register-transfer level testability analysis and optimization," *IEEE Trans. VLSI Systems*, vol. 9, pp. 824–832, Dec. 2001.

[11] L. Lingappan and N. K. Jha, "Unsatisfiability based efficient design for testability solution for register-transfer level circuits," in *Proc. VLSI Test Symp.*, pp. 418–423, May 2005.

[12] L. Lingappan, S. Ravi, and N. K. Jha, "Satisfiability-based test generation for nonseparable RTL controller-datapath circuits," *IEEE Trans. Computer-Aided Design*, vol. 25, pp. 544–557, Mar. 2006.

[13] B. T. Murray and J. P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 594–603, June 1990.

[14] H. K. Lee and D. S. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1048–1058, Sept. 1996.

[15] I. Ghosh, A. Raghunathan, and N. K. Jha, "Hierarchical test generation and design for testability methods for ASPPs and ASIPs," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 357–370, Mar. 1999.

[16] *Design Compiler*. Synopsys Inc. (`http://www.synopsys.com`).

[17] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*. Cambridge University Press, 1993.

[18] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, New York, 1993.