Efficient Sequential ATPG for Functional RTL Circuits *

Liang Zhang[†], Indradeep Ghosh[‡], and Michael Hsiao[†]
[†] Department of ECE, Virginia Tech, Blacksburg, VA, {liang, hsiao}@vt.edu
[‡] Fujitsu Labs. of America Inc., Sunnyvale, CA, ighosh@fla.fujitsu.com

Abstract

We present an efficient register-transfer level automatic test pattern generation (ATPG) algorithm. First, our ATPG generates a series of sequential justification and propagation paths for each RTL primitive via a deterministic branch-and-bound search process, called a test environment. Then the pre-computed test vectors for the RTL primitives are plugged into the generated test environments to form gate-level test vectors. We augment a 9-valued algebra to efficiently represent the justification and propagation objectives at the RT Level. Our ATPG automatically extracts any finite state machine (FSM) from the circuit, constructs the state transition graph (STG), and uses high-level information to guide the search process. We propose new static methods to identify embedded counter structures, and we use implication-based techniques and static learning to find the FSM traversal sequences sufficient to control the counters. Finally, a simulation-based RTL extension is added to augment the deterministic test set in a few cases when there is additional room for the improvement in fault coverage. Experimental results show that our new deterministic RTL techniques achieve several orders of magnitude reduction of test generation time without compromising fault coverage when compared to gate-level ATPG tools. Our ATPG also outperforms a recently reported simulation-based high-level ATPG tool in terms of both fault coverage and CPU time.

1. Introduction

Deep sub-micron technology is allowing designers to add significantly more transistors to a single chip, and the corresponding manufacturing process is becoming more complex. Because of the exponential complexity of sequential ATPG, its performance is extremely sensitive to the size of the circuit. Thus, conventional gate-level sequential ATPG methods frequently produce unsatisfactory results due to the large circuit sizes. On the other hand, the functional register-transfer-level (RTL) ATPG has the advantages of fewer circuit primitives, easier access to circuit functional information (abstraction of block functionality), and earlier application of ATPG to a higher level during the design cycle.

*This research was supported in part by a grant from Fujitsu Labs of America, and in part by NSF Grants CCR-0196470.

Design for testability (DFT) techniques, such as scan and built-in self-test (BIST), are widely used in practice to reduce the complexity of sequential ATPG. DFT essentially augments the circuits with additional logic that is active only in the test mode. The trade-offs of DFT with reduced ATPG complexity are the overhead of chip real estate, potential performance degradation, and significantly longer test application time for scan-based approach.

In the past, many different algorithms have been proposed for sequential ATPG at different levels of circuit abstraction. At the gate level, both deterministic [11, 4] and simulation-based [10, 9] approaches have been proposed. At the RT level, ARTIST [3], a genetic algorithm (GA) based ATPG, targets the statement coverage of behavioral HDL descriptions. Also at the RT level, the authors presented in [7] an ATPG based on the comparison between the error-free and erroneous BDD representations of the VHDL specification. The bit coverage was used as the error model in that framework. In [8], the authors proposed a deterministic functional RTL ATPG algorithm which utilizes a set of 9-valued algebra to perform symbolic justification and propagation of test objectives. TAO, presented in [12], is a regular expression based RTL ATPG, which involves writing path equations for modules. There are also some mixed-level ATPGs such as those reported in [1, 14, 13]. In [1], the test generation is split into three steps, combinational test generation, fault-free state justification, and fault-free state differentiation. The combinational test generation works on gate-level circuit description, while the other two steps are performed on RTL circuit. As the authors mentioned, this approach works well on circuits with highly connected state transition graphs (STGs). In [14], the authors first manually generate sequences which cover all statements in the RT-level description, then those sequences are fed to a gate-level ATPG to generate gate-level test vectors. The algorithm in [13] first performs ATPG on RT-level circuits, then performs fault simulation to drop all tested faults, finally uses gate-level ATPG to target remaining faults. Our approach is *different* from both gate-level or mixed-level ATPGs in that it requires no gate-level description of circuits, thus it can be applied early in the design cycle.

In this paper, we try to address the sequential ATPG problem at the functional RT level without any help from DFT. The input circuit to our ATPG is synthesizable VHDL or Verilog code. We have developed a preprocessor to both improve the

efficiency and also to learn high-level information (such as extracting state transition graphs, identifying embedded counters and its excitation sequence, static learning, etc.) from RTL circuits. Based on the RTL ATPG [8] as our basic algorithm, our ATPG uses a 9-valued algebra to efficiently generate test environments for each RTL primitive. The ATPG then translates the test environment into the system-level test vectors by plugging the pre-computed test vectors into it. On top of the basic algorithm, the 9-valued RTL algebra was extended to 10-valued algebra with new symbol *Cp* and the *value range*, so that the ATPG can handle control-intensive circuits more effectively. To speed up the value range analysis, we incorporated new RT-level implication-based techniques into ATPG framework. Static learning has been applied to collect the high-level information of the circuit, which are used to guide the ATPG search process. All the above techniques are integrated into our ATPG algorithm at the RT level called High-level Test (HTest) generator.

HTest achieves the same as or better fault coverage than HITEC [11], a deterministic gate-level sequential ATPG, for all except very few circuits, while reducing the execution time up to *four orders of magnitude* on sequential benchmark circuits. For those few circuits where HTest lags behind gate-level ATPGs, we give an analysis and discussion on the case to why high-level ATPGs can potentially suffer. To remedy this, a RT-level simulation-based approach can be used to enhance the coverage to take care of the special case scenario. Compared to other high-level ATPGs, HTest outperforms a recently reported genetic algorithm (GA) based behavioral ATPG in terms of both fault coverage and test generation time. Finally, because our framework targets RTL circuits, it can also produce effective vectors for validation purposes at the RT level.

The rest of the paper is organized as follows. Section 2 discusses the preliminaries and background for RTL ATPG, Section 3 describes our ATPG algorithm, including augmentation of original RTL algebra, and the techniques to speed up the ATPG process. Section 4 reports and discusses the experimental results, and finally Section 5 concludes the paper.

2. Preliminaries

2.1. Assignment Decision Diagrams

First introduced in [2] for high-level synthesis, an assignment decision diagram (ADD) is a compact representation for functional RTL circuits. ADDs can be viewed as a set of control switches, which select the proper inputs to feed target nodes. ADDs also provide us a structural view of the circuit under test (CUT), which makes ADD a good candidate representation for testing purposes. Figure 1 shows a segment of VHDL code, and part of the associated ADD. The VHDL code contains a counter, and only the part of ADD associated with the counter is shown. The triangular-shaped symbol is an Assignment Decision Node (ADN). The binary control signals (CTL1 and CTL2) feeding into the ADN are mutually exclusive. In this case, depending on the values of CTL1 and CTL2 signals, the ADN will select the corresponding value to be routed to the output. The other circular nodes are simply arithmetic or logical operators that comprise the computing paths in the circuit.

VHDL Expression:

```
if (RESET = '0') then
    STATE := S0;
    Reg <= 0 ;
elsif Clock'event and Clock = '1' then
    if STATE = S0 then
        if E1 then
            STATE := S1:
            if E2 then
                Reg <= Reg + 1;
            end if;
        else
            STATE := S0;
        end if;
    end if;
end if;
```

ADD Representation for Counter:

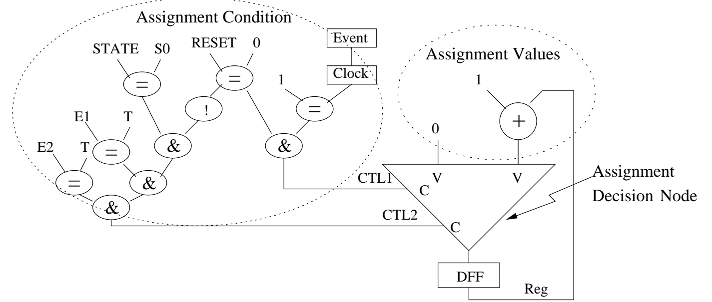

Figure 1. Assignment Decision Diagram

2.2. RTL Algebra Based Testing Approach

In [8], the authors defined following 9 symbols for RTL justification and propagation purposes.

- **Cg** (general controllability) is the ability to control its value to arbitrary value.
- **C0** (controllability to zero) is the ability to control the variable to the value 0.
- **C1** (controllability to one) is the ability to control the variable t0 1; i.e., "000...01".
- **Ca1** (controllability to all ones) is the ability to control the variable to all ones; i.e., "111...11".
- **Cq** (controllability to a constant) is the ability to control the variable to any fixed constant.
- **Cz** (controllability to the Z value) is the ability to control the variable to high-impedance Z.
- **Cs** (controllability to a state) is the ability to control the state variable to a particular state.
- **O** (observability) is the ability to observe a fault at a variable.
- **O'** (complement observability) is defined for single-bit variables only. It signifies the zero/one fault.

During the testing of each RTL primitive, symbolic RTL justification and propagation are performed using a branch-and-bound search procedure to trace out the path, which starts from the primary inputs (PIs), traverses through the target site, and continues to one or more primary outputs (POs). The obtained symbolic path, which may span across multiple clock cycles, is termed a *test environment*. Then each pre-computed test vector from the test set library for the module under test (MUT) can be readily plugged into the test environment to obtain the system-level test set for the RTL module.

3. Our ATPG Algorithm

Our ATPG framework consists of a front-end parser, a pre-processor, and a two-phase testing procedure. First, front-end parser reads in the synthesizable VHDL/Verilog circuit and converts it to a hierarchical netlist of ADDs. Then, a pre-processor is invoked, which flattens the netlist of ADDs, performs static learning, extracts FSM information, and computes the controllability and observability measures.

The first phase of testing procedure uses a deterministic engine, which derives the test environment for each MUT. The targeted MUTs are arithmetic operations, logic arrays, ADN nodes, random logic, and black boxes. Whenever a test environment is found, the second phase of testing procedure is invoked to plug the pre-computed test vectors for the given primitive into the test environment to get system-level test vectors. Finally, all test vectors are concatenated and written to user specified file.

Section 3.1 and 3.2 describe our implementation of the algebra-based RTL ATPG. Subsequent subsections detail the new techniques incorporated to the framework.

3.1. Test Environment Generation

The test environment can also be viewed as a set of conditions that allow controllability and observability of the target RTL primitive. The test environment generation process is essentially searching for a sufficient symbolic path, through which the excitation objectives can be delivered to the target site, and error effect can be propagated to the PO.

Similar to the high-level approach in [8], our ATPG goes through the following steps to generate the test environment for each RTL primitive in the target list.

1. Find a symbolic propagation path, which starts from the target and ends at a primary output (PO). If either no path exists or all paths have been tried, exit for the current target.
2. Inject algebra-based constraint objectives on the side-inputs of this path so that the erroneous response at the output of target can be propagated to a PO.
3. Inject algebra-based excitation objectives at the inputs of target.
4. Apply the implications, if any, to early detect conflicts.
5. Justify all objectives via a branch-and-bound procedure.
6. If all objectives are justified, generate a test environment, then exit. Otherwise, try a different propagation path for the target and go to step 1.

Consider the following VHDL code segment as an example to illustrate the above algorithm. Let's assume that *RST*, *ina* and *inb* are PIs, and *out* is the PO for the circuit.

```
if RST = '1' then
  a:=1; b:=1; c:=0; out:= 0; i:=0;
elsif clk'event and clk = '1' then
  i:= i+1;
  if (i=1) then
     a := ina + 3;
     b := inb;
  elsif (i=2) then
     c := a * b;
  elsif (i=3) then
     out := c;
  endif;
endif;
```

The compiled structural RTL representation is shown in Figure 2. Suppose that the multiplier M3 is our current target. Our algorithm will execute as follows:

1. Find the shortest propagation path $M3 \rightarrow M4 \rightarrow C \rightarrow M5 \rightarrow OUT$.
2. Propagation constraints of (CTL1,1,C1) and (CTL2,2,C1) are injected. The first objective of (CTL1,1,C1) means a C1 algebra at time frame 1 is needed on signal CTL1.
3. Excitation objectives of (a,0,Cg) and (b,0,Cg) are injected at the inputs of M1. The objectives mean algebra Cg is needed on both signal a and b at time frame 0.
4. Justify all objectives via a branch-and-bound search.
5. All objectives are justified. The test environment is generated as shown below. Note that the test environment contains all justified algebra on the PIs and Cgs on internal nodes. The fan-in nodes are also included in test environment for the internal nodes.

The generated test environment can be visualized as following:

```
(RST,-2,C1);
(RST,-1,C0);
(ina,-1,Cg);
(M0 ,-1,Cg):  (ina,-1,Cg);
(M1 , 0,Cg):  (M0 ,-1,Cg);
(a  , 0,Cg):  (M1 , 0,Cg);
(inb,-1,Cg);
(M2 , 0,Cg):  (inb,-1,Cg);
```

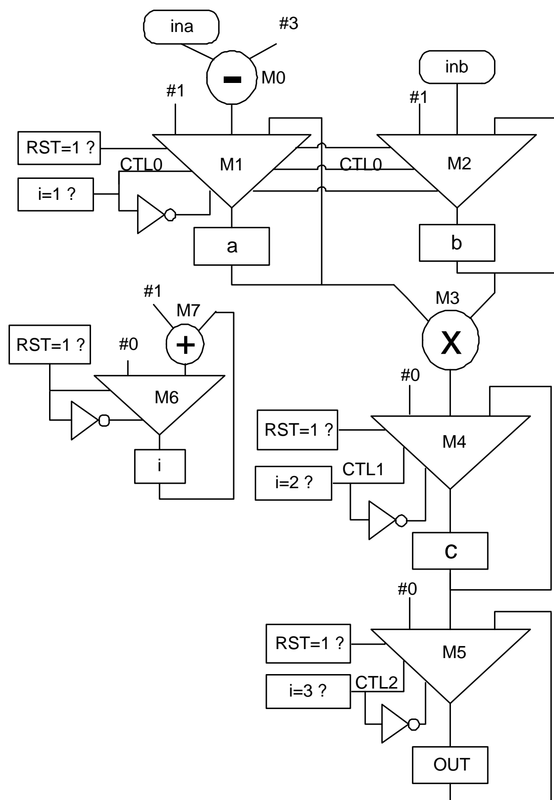

Figure 2. Structural RTL for the VHDL code

```
(b  , 0,Cg):  (M2 , 0,Cg);
(RST, 0,C0);
(RST, 1,C0);
(RST, 2,C0);
```

In the next section, we will show how to translate the test environment into system-level vectors.

3.2. Test Environment Translation

A generated test environment must be translated into system-level vector(s) to be applicable to the circuit. Consider the test environment generated previously in Section 3.1 as an example to illustrate the translation procedure.

Table 1 highlights partial results as the procedure proceeds. Suppose we need to apply 5 and 10 at the inputs of the multiplier M3 respectively. First, all algebra except the *Cg* at PIs are translated. For this example, only the value of RST is determined at this step, as shown from the columns under the heading "STEP 1".

Secondly, the value 5 is plugged into (a,0), then following the trace $(a, 0, Cg) \rightarrow (M1, 0, Cg) \rightarrow (M0, -1, Cg) \rightarrow (M0, -1, Cg) \rightarrow (ina, -1, Cg)$ the value can be propagated backward to the PI. The value may need to be adjusted when propagated through certain types of RTL constructs. For example, the value 5 is propagated from (a,0) through (M0,-1) without any adjustment. However the 5 on (M0,-1) implies the 8 at (ina,-1), since M0 is a subtracter, and the other operand is constant 3. Note that, the information of primitive type of M0 and the other input operand of M0 are not directly stored in the test environment. Instead, they can be quickly retrieved from the circuit netlist. Similarly the 10 can be plugged in (b,0,Cg) and the value can be propagated to the (inb,-1). The translation results are recorded in columns under "STEP 2".

Finally, all unspecified PIs are filled with the random numbers to form the fully specified test vectors. The last three columns show the final test vectors.

Table 1. Test environment translation

Time Frame	STEP1			STEP2			STEP3		
	RST	ina	inb	RST	ina	inb	RST	ina	inb
-2	1	x	x	1	x	x	1	2	3
-1	0	x	x	0	8	10	0	8	10
0	0	x	x	0	x	x	0	1	0
1	0	x	x	0	x	x	0	4	7
2	0	x	x	0	x	x	0	5	5

3.3. Augmenting the Algebra with Value Range

As demonstrated in the previous example, once we generate a test environment, we can easily deliver test vectors to the MUT. However, the inadequacy of the original 9-valued algebra may hinder the test environment generation process. Figure 3 shows a segment of VHDL code and its corresponding ADD representation. Suppose signal *a* is on the propagation path of test environment. Then the signal *En* must be asserted to C1 to select the output of subtracter to *Dataout*. To justify the C1 on *En* , the *Cn1* needs to be C1, which leads to the the general controllability of Cg on signal *Data*. At the same time, the *Cn2* needs to be C0, which requires another Cg on the *Data*. Since no signal can be arbitrarily set to two different values simultaneously, two Cgs lead to the unresolvable conflict. With the original 9-valued algebra, there is no other way to justify the propagation conditions. As a result, the ATPG fails to generate corresponding test environment and subsequently aborts. However, we can see that (En,C1) is actually justifiable. With any value between 50 and 59 of *Data*, the *En* is justified to C1.

The reason that the previous ATPG does not find the solution is that in justifying the (Cn1, C1), *Cg* on *Data* is too strong of a condition. However, with the original 9-valued algebra, *Cg* is the only possible value.

To overcome this inadequacy, we introduce a new symbol **Cp** with a value range to the RTL algebra set. A justified Cp together with a specified value range, say [30;0], on a signal *X* means that we can control the *X* to be a value between 30 and 0. But we do not have to control which particular value *X* should be.

In the above example, to justify the (Cn1, C1), we only need to justify (Data, Cp[MAX;50]), since we do not care the exact value of Data as long as it is greater or equal to 50. (MAX is the largest number representable by Data.) Similarly, the (Data, Cp[59;0]) satisfies the (Cn2,C0). Therefore, we are able to relax

```
S1:     If (Data > 59) then
S2:         Dataout := 15;
S3:     elsif (Data > 49) then
S4:         Dataout := a - b;
S5:     endif;
```

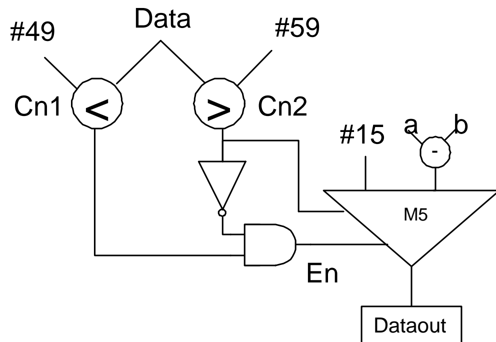

Figure 3. Augmenting the algebra with value range

the previously strong condition to a much weaker yet sufficient condition. To simultaneously satisfy both Cps, the combined justification objective becomes (Data, Cp[59;50]).

We can see that the resultant justification objectives are exactly the conditions expressed at VHDL statement *S3* and *S1*. Note that two Cps on the same signal are not compatible if and only if the two Cps have non-overlapping value ranges.

The augmentation of the new symbol and the related value range analysis increase the computation overhead; however, they are indispensable in modeling the justification objective more accurately. As a result, they are used sparingly only when the initial algebra fails to generate a test environment.

3.4. Static Learning to Aid Value Range Analysis

In the previous section, we demonstrated the necessity of inclusion of value range analysis to enhance the symbolic justification. However, there is a related performance issue.

In that example, to justify the (En, C1), both (Cn1, C1) and (Cn2, C0) are needed. Suppose ATPG picks (Cn2,C0) first, which leads to the (Data, Cp[59;0]). Since the justification is conducted in a depth-first fashion, the ATPG goes on to justify (Data, Cp[59;0]) immediately. Suppose after a certain number of steps, the (Data, Cp[59;0]) is successfully justified. Then, ATPG moves on to justify (Cn1, C1), which leads to a tighter value rang of [59;50] on *Data*. This tighter value range needs to be justified as well. Therefore, the ATPG has to justify the two related Cps on the *Data* respectively. For some circuits, this only increases the computation moderately if the two Cps are both easy to be justified. However, for some other circuits, the first unnecessary justification can cause large number of backtracks. For example, the value of Data is fed from the memory as shown in Figure 4. With the first objective Cp[59;0], the

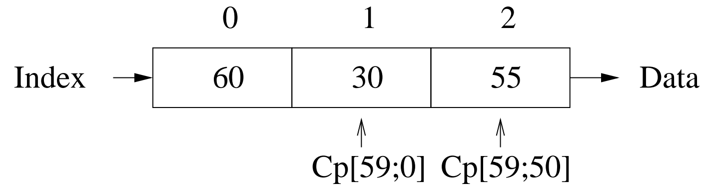

Figure 4. Inaccurate value range selects wrong signal value.

value 30 may be selected, and accordingly the *index* is justified to 1. However, the selected value of 30 conflicts with the second objective Cp[59;50]. If the justification of *index* is not trivial, a large number of backtracks will be needed before ATPG eventually picks correct value 50 if given enough time.

In order to quickly determine the desirable value range for each encountered objective, we use static learning process to construct the RTL implications. It is desirable to have a mechanism that can determine the value range as tight as possible and as quickly as possible.

We augmented HTest with following static learning steps to find RTL implications.

1. For each comparator type RTL primitive (=, /=, >, <, and etc), inject two implication constructs as in Figure 5. The upper bound of value range is determined by the signal bit width.

2. Recursively forward propagate the implication constructs using the rules shown in Figure 5. We only propagate through binary logic gates, such as BUF, INV, AND, NAND, OR, NOR. Implication propagation stops at all other gates.

3. Update the implication list for reached RTL primitives.

By only injecting the implications at comparator primitives and by only propagating through binary logic gates, our implication process focuses on the control logic of circuit, since our algebra-based ATPG has been proved to be effective for datapath circuits.

Following the same example in Figure 3, after the static learning, we have two implications on *En*, which are C1→(Data,Cp[MAX;50]), and C1→(Data,Cp[59;0]). When justifying algebra C1 on En, checking the implication lists, ATPG knows that two implications are required. Immediately, the ATPG can derive correct value range of [59;50] for the signal *Data*. Thus, unnecessary steps and backtracks are avoided.

Our seemingly simple implication procedure is actually very powerful in finding implications for the control logic and in avoiding conflicts. First, the control inputs of ADN are mutually exclusive; that is, any control signal being C1 implies all other control signals feeding the same ADN are C0. Therefore C1 on any control input will have many implications. Therefore using statically computed implication can speed up the justification process. Secondly, asserting any control input of ADN

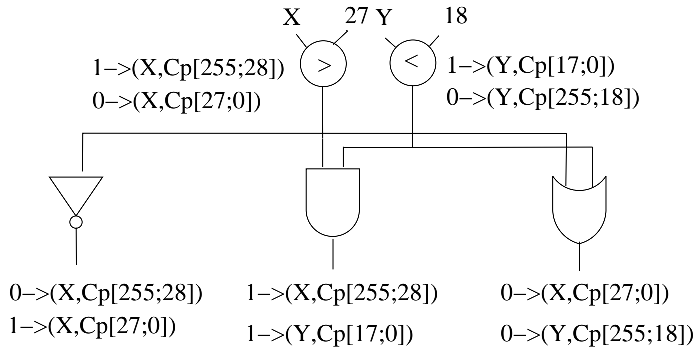

Figure 5. Direct implication of RTL algebra

is functionally equivalent to specifying a particular basic block in the HDL code to be executed. Even though the execution path reaching that basic block may not be unique, it is very likely that a particular code segment is guaranteed to be executed. Therefore, the related condition variables are implied by the single assertion of control input on ADN. Thirdly, the nested conditional statements in VHDL/Verilog always induce the re-convergent predicate signals. Therefore, there are many opportunities to apply this implication-based techniques.

3.5. STG Extraction

State justification is known as one of the most difficult problems in sequential ATPG. Unlike gate-level techniques, which cannot view the global relationships among flip-flops, our HTest can readily extract the STG directly from the RTL netlist. For each FSM inside the circuit, there is one ADD associated with the state variable of the FSM. At RT level, all states of a FSM are explicitly declared. Thus, the size of states is always manageable. The complete STG of FSM can be easily constructed.

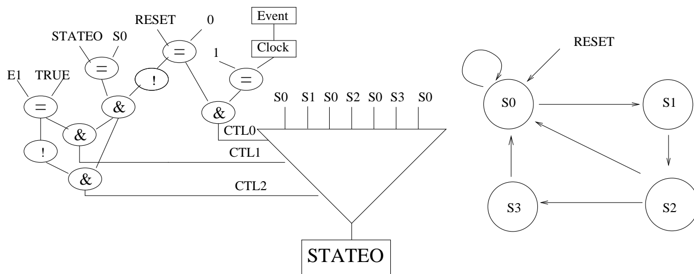

Figure 6. ADD for state variable & STG

Figure 6 shows the ADD for the state variable *STATEO*, and extracted STG. The corresponding VHDL code is shown in Figure 1. Note that, we do not draw out the complete control logic for the ADD. Each control input except the *RESET* is associated with a single FSM state. For example, the control inputs CTL0, CTL1, and CTL2 are all controlled by *state S0*. The extracted STG is complete in the sense that it contains all the states of the FSM, and captures all transition arches of the FSM. However, it does not contain the transition conditions associated with each arch. The transition conditions are implicitly embedded in the circuit. Every FSM has a reset state, which is S0 in this example. Conceptually, every state is the predecessor state of reset state. However, we do not store resetting arches in STG, since the resetting arches are not favored for test generation purposes. Instead, we just label the reset state. Implicitly, every state can go to reset state in one step.

3.6. Counter and Iteration Loop Identification

Counters are difficult for ATPG, especially when the counters require long sequences. In some circuits, counters may even be the bottleneck for testing. For example, the output of a counter may be used to control the input of another module. With counters frequently used in modern digital systems, having an engine that can automatically extract and exploit this knowledge would be of immense aid. The algorithm in [8] proposed a heuristic for counter identification. The approach dynamically checks for the repetitive appearances of the same justification objective on the same variable across a number of time frames. One drawback of that approach is high computation overhead. It needs to check the condition frequently throughout justification process. Secondly, the counter identification can be a false positive. In other words, it may mistakenly declare a non-counter structure to be a counter.

The counter identification in our HTest is performed statically. In the structural RTL representation, the counter structure contains an adder, a feed-back loop through some flip-flops and ADN, and a constant input to the adder, as shown in Figure 1. During the preprocessing step, our algorithm marks all adder modules that fit in the counter structure, then perform static learning for each counter.

Consider again the counter example in Figure 1 to illustrate how our ATPG identifies the FSM iteration loop for the counter. The FSM iteration loop is the shortest state sequence loop that increments the counter continuously. Note that, the ADD for state variable and the STG are shown in Figure 6. For example, inspecting the VHDL code and STG we can see that $S0 \rightarrow S1 \rightarrow S2 \rightarrow S0$ is the shortest FSM loop, which increments the counter continuously. Suppose during the justification, we need to justify the counter to be 31. Then, the shortest state sequence is ($S0 \rightarrow S1 \rightarrow S2$)x31, a total 62 states!. Without this knowledge, the branch-and-bound method often searches blindly in the large search space.

Our ATPG uses the following steps to identify the above sequence loop.

1. Perform breath-first search on circuit netlist at the control input of counter ADN to find out the controlling state. That state is the operation state for the counter. In this example, S0 is controlling the CTL2. Thus S0 is the operation state of counter.

2. Assert the control input of ADN and compute the implication of it. For example, the assertion of CTL2 implies E2

is true, E1 is true, and etc.

3. Check state variable ADN (in Figure 6) to see if any control input is implied to C1. The state at the corresponding data-input is the next state. In this example, CTL1 is implied to C1, therefore the next state is S1.

4. Do breath-first search on extracted STG to find the shortest path from counter next state to counter operation state. Note that, the reset arches are not allowed. Here we get $S1->S2->S0$.

5. The final iteration loop contains 3 states and 3 transition arches found in previous steps. $S0 \rightarrow S1 \rightarrow S2 \rightarrow S0$.

The above implication-based approach is performed statically during the preprocessing step. The identified FSM iteration loop for each counter is stored in a data structure and will be used during the justification process to quickly guide the ATPG to the correct solution.

3.7. Optional Simulation-based Procedure for Special Case

Figure 7 shows one of the cases that RTL algebra cannot effectively handle, even with the the added value analysis extension. In this example, signal b needs to be justified to ”11010111”. Even though we can model the justification objective as (Cp,[215;215]), this justification objective cannot be justified successfully through signal *a* due to the bus re-joint. It is difficult to represent the justification objective collectively with just one algebraic symbol. There are too many possible combinations for a[15:8] and a[7:0] to justify the *b*. As a result, the possible values of a[15:0] are scattered and cannot be efficiently represented by RTL algebra. Therefore backward justification aborts on this. For this particular case, gate-level ATPG will potentially be more effective and efficient than RTL ATPG. In general, when word-level entities are combined through Boolean operators, the resulting value ranges will be difficult to obtain. If such situation arises, HTest will likely be unable to effectively generate test environment for the affected RTL primitives. As the result, the HTest may suffer in fault coverage.

On the other hand, the simulation-based approach performs the forward simulation only. It may be able derive the vectors satisfying the conditions on *b*. In our framework, we incorporated a genetic algorithm (GA) based ATPG procedure to boost the fault coverage when deterministic test generation aborts on the primitives similar to the one discussed above. As will be shown, this extension of simulation-based engine only needs to be applied to very few circuits. We use the observability-based code coverage (OBCC) [5] as the coverage metrics for the GA process. OBCC coverage is superior to the classical software testing coverage (such as statement and branch coverage) in that it incorporates observability as well as controllability information into the simulation. For the untested RTL primitives, the GA process is invoked to generate vectors, that excite the primitives and bring the output primitives to the observable points.

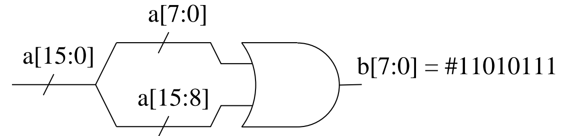

Figure 7. Bus split and re-joint

4. Experimental Results

The original prototype of [8] was implemented with TCL/TK script. We implemented the original algorithm in C++ and use it as the base-line of our experiment. HTest was implemented on top of the base-line with the proposed new techniques, totaling about 12,000 lines of C++ code. Experiments [8] have demonstrated that original RTL ATPG is very efficient for data-path circuits with large RTL-to-gate expansion ratios ($number\ gates/number\ VHDL\ lines$).

In our experiments, we used 12 sequential benchmark circuits including data-path circuits as well as control intensive circuits. 10 circuits are from the ITC99 benchmark suite, *Paulin* is from [8], and *GPIO* is an industrial general-purpose input-output bus controller.

The benchmark circuit characteristics are listed in Table 2. Each circuit has an explicit reset input. For each circuit, the number of VHDL lines are given, the size of the gate-level circuits (# gates and # FFs) are also reported. It can be seen that in the circuit *Paulin* the RTL-to-gate expansion ratio is very high.

Table 2. Benchmark Circuits Characteristics

Circuit	RTL	Gate level	
	VHDL Lines	# Gates	# Flip-flops
B01	110	56	5
B02	70	30	4
B03	141	181	30
B04	102	547	66
B05	332	643	34
B06	128	76	9
B07	92	434	51
B08	89	190	21
B10	167	190	17
B11	118	397	30
Paulin	130	39558	227
GPIO	1002	1720	148

We compared the test generation results of HTest with HITEC[11], STRATEGATE[10], and ARTIST[3]. HITEC is a deterministic gate-level sequential ATPG tool, STRATEGATE is a state-of-the-art simulation-based gate-level sequential ATPG tool, and ARTIST is a recently reported behavioral RTL ATPG tool. In [6], the authors of ARTIST reported a newer behavioral level ATPG, PRINCE, which achieves slightly higher fault coverage than ARTIST, but with the test generation time reported as ”from some hours to two days”. Therefore, we only compared our results with ARTIST, since our execution times are extremely short. To demonstrate the effectiveness of

our approach, we turned off the simulation-based procedure for all circuits except the B08. The results are reported in Table 3. For each circuit, the gate-level fault coverage (FC), execution time (TGen) in CPU time, and the test set size (TSize) are reported. The platform for HITEC, STRATEGATE and HTest was an Ultra 5 330MHz workstation with 128 MB Memory, while the results of ARTIST were obtained on a Ultra-5 330MHz with 256 MB memory. The gate-level fault coverage of HTest is computed by fault simulating the test vectors HTest derived on the corresponding gate-level circuits.

From this table, we can see that HTest outperforms ARTIST for all circuits in terms of both fault coverage and test generation time. For example, in circuit B05, ARTIST obtained a fault coverage of 33.5% in 33,393 seconds, while our HTest achieved 48.0% in only 19.56 seconds. Compared to HITEC and STRATEGATE, HTest achieved orders of magnitude reduction of test generation time without compromise in fault coverage. For the circuit *Paulin*, because it has a very high RTL-to-gate expansion ratio, the advantage of fewer primitives at the RT level over gate level is most significant, where HTest achieves 99.73% FC (higher than both HITEC and STRATEGATE) in only 1.8 seconds, while HITEC took a couple days of execution and STRATEGATE over a day. Similarly, in B05, the execution time of HTest was also significantly shorter. The reported test results of circuit B08 are obtained first with the deterministic approach, followed with the simulation-based engine turned on. The deterministic approach alone achieves the fault coverage of 89.6% for B08 within 4.39 seconds of CPU time, which is already higher than that reported by the simulation-based ARTIST. The simulation-engine increased the fault coverage further to 98.9%, in just less than 3 additional seconds.

Note that, a previous RTL ATPG [8] reported better ATPG results than gate-level ATPG only for circuits with RTL-to-gate expansion ratios of 5 or more. However, the results of HTest are comparable or better than that of gate-level ATPG for *any* expansion ratio. In terms of test set sizes, HTest generated more vectors than HITEC for all circuits. This is due to the fact that HTest does not use any specific RTL fault model, thus no fault simulation and no fault dropping can be performed to reduce the test size. Additionally, without specific fault model, HTest has to target each RTL primitive while HITEC only needs to target each fault on the collapsed gate-level fault list. For some circuits, HTest generates fewer vectors than STRATEGATE. Compared to ARTIST, HTest produces similar number of vectors for the circuits with comparable fault coverages. To overcome the problem of large test set sizes gate-level test compaction techniques may be used as a post process after RTL test generation. This is an effective and inexpensive technique and for our examples, it is able to reduce all HTest test sets to below that of gate-level test generators in very little CPU time.

Figure 8 shows the fault coverages for HTest (without simulation-based procedure) and our implementation of baseline RTL ATPG of [8] (circuits with the same fault coverages are not shown). The effectiveness of our new techniques is best demonstrated with B05, B07, and B08, for which the baseline ATPG aborts on most of RTL primitives and only generates very few test environments. As a result, large number of RTL primitives remain untested. As per the figure, applying the techniques presented in this paper, HTest is able to generate much more test environments and thus to improve the fault coverages dramatically. Test generation time is also reduced in HTest. For example, the original RTL ATPG implementation takes 5543 seconds to achieve 93.5% fault coverage for *GPIO*, and 138 seconds to achieve 99.72% fault coverage for *Paulin*.

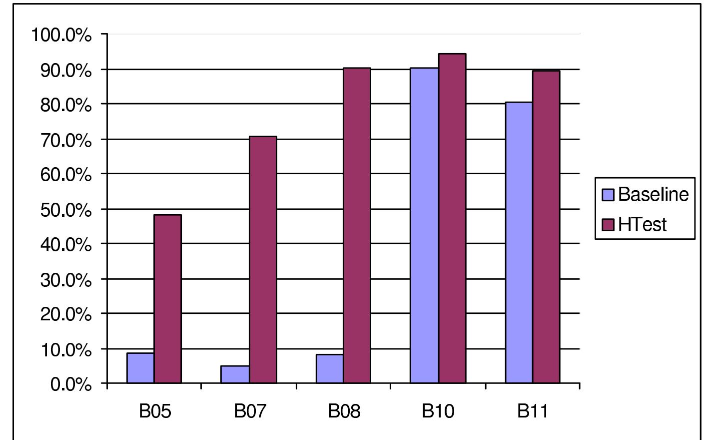

Figure 8. Baseline RTL ATPG vs. HTest

5. Conclusion

We have presented an efficient sequential ATPG at the register-transfer level. Whenever the deterministic test environment generator is successful in generating the test environment for the target RTL primitive, gate-level test vectors can be directly derived from the corresponding environments by plugging in the pre-computed test vectors for that RTL primitive. Our ATPG is able to automatically extract state transition graphs, identify any embedded counters, and perform high-level static learning. We also propose an implication-based technique to efficiently handle the interaction of counters and FSM. In addition, we augmented the previously proposed algebra-based ATPG approach with value range analysis ability, which is crucial to the RTL circuits with nested conditional statements. The experiments demonstrate that our approach is able to generate high quality test sets with extremely low CPU times for all types of designs. For the circuits with high RTL-to-gate expansion ratio, several orders of magnitude reduction in computation over existing techniques is observed.

Finally, the fact that our approach requires only RTL description is very valuable in that it can be applied early in design cycle, and can be used for design validation purposes.

References

[1] A. R. N. Abhijit Ghosh, Srinivas Devadas. Sequential test generation and synthesis for testability at the register-transfer and

Table 3. Test Generation Results

Circuit	STRATEGATE			HITEC			ARTIST			HTest		
	FC (%)	TGen (sec)	TSize	FC (%)	TGen (sec)	TSize	FC (%)	TGen (sec)	TSize	FC (%)	TGen (sec)	TSize
B01	**100**	1.55	86	**100**	0.2	138	**100**	4118	1061	**100**	0.06	811
B02	**100**	0.71	85	**100**	0.07	62	99.33	1731	940	**100**	0.03	298
B03	79.1	1752	489	77.9	245	217	74.33	5131	374	**78.0**	3.68	814
B04	**90.8**	616	9042	88.5	225	293	89.42	6905	427	*89.6*	0.12	768
B05	**48.0**	99544	7584	**48.0**	1700	225	33.50	33393	2800	**48.0**	19.56	23241
B06	**97.9**	1.43	59	**97.9**	0.25	87	97.02	2315	62	**97.9**	0.06	535
B07	70.5	8604	6757	69.2	892	50	57.53	2251	461	**70.8**	5.97	538
B08	**99.5**	232	1566	**99.5**	24	593	86.27	2106	329	*98.9*	7.32	1749
B10	**96.4**	274	1308	*94.4*	56.4	301	90.42	10851	586	*94.4*	17.58	2847
B11	89.1	7849	4430	74.4	677	151	85.98	5092	532	**89.4**	5.85	972
Paulin	**99.7**	101071	4499	97.9	147002	752	-	-	-	**99.7**	1.81	2093
GPIO	98.3	11078	5292	**99.4**	57	1560	-	-	-	97.6	96.1	38247

No simulation-based procedure applied in HTest except for B08.

logic levels. *IEEE Trans. Computer-Aided Design*, 12(5):579–588, May 1993.

[2] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *Proc. Design Automation Conf.*, pages 413–418, June 1993.

[3] F. Corno, M. S. Reorda, and G. Squillero. RT-Level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers*, 17(3):44–53, July-September 2000.

[4] H.-K. T. M. S. Devadas, A. R. Newton, and A. Sangiovanni-vincentelli. Test generation for sequential circuits. *IEEE Trans. Computer-Aided Design*, 7(10):1081–1093, October 1988.

[5] F. Fallah, S. Devadas, and K. Keutzer. OCCOM – Efficient computation of observability-based code coverage metrics for functional verification. *IEEE Trans. Computer-Aided Design*, 20(8):1003–1015, August 2001.

[6] F.Corno, G.Cumani, M. Reorda, and G.Squillero. Effective techniques for high-level ATPG. In *Proc. IEEE Asian Test Symposium*, pages 225–230, 2001.

[7] F. Ferrandi, F. Fummi, and D. Sciuto. Implicit test generation for behavioral VHDL models. In *Proc. International Test Conference*, pages 436–441, 1998.

[8] I. Ghosh and M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Trans. Computer-Aided Design*, 20(3):402–415, March 2001.

[9] R. Guo, S. Reddy, and I. Pomeranz. PROPTEST: a preperty based test pattern generator for sequential circuits using test compaction. In *Proc. Design Automation Conference*, pages 653–659, 1999.

[10] M. Hsiao, E. Rudnick, and J. Patel. Sequential circuit test generation using dynamic state traversal. In *Proc. Eur. Design Test Conf.*, pages 22–28, Mar 1997.

[11] T. Niermann and J. Patel. HITEC: A test generation package for sequential circuits. In *Proc. Eur. Design Automation Conf.*, pages 214–218, Feb. 1991.

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

[13] S. Ravi and N. Jha. Fast test generation for circuits with RTL and gate-level views. In *Proc. International Test Conference*, pages 1068–1077, 2001.

[14] E. M. Rudnick, R. Vietti, and et al. Fast sequential circuit test generation using high-level and gate-level techniques. In *Proc. Design Automation and Test in Europe*, pages 570–576, 1998.