Scalable Test Generation by Interleaving Concrete and Symbolic Execution

Xiaoke Qin and Prabhat Mishra

Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA
{*xqin, prabhat*}@*cise.ufl.edu*

***Abstract*—Functional validation is widely acknowledged as a major challenge for System-on-Chip (SoC) designs. Directed tests are superior compared to random tests since a significantly less number of directed tests can achieve the same coverage goal. Existing test generation techniques have inherent limitations due to use of formal methods. First, these approaches expect formal specification and do not directly support Hardware Description Language (HDL) models. Most importantly, the complexity of real world designs usually exceeds the capacity of model checking tools. In this paper, we propose a scalable technique to enable directed test generation for HDL models by combining static analysis and simulation based validation. Unlike existing approaches that support a limited set of HDL features, our approach covers a wide variety of features including dynamic array references. We have compared our approach with existing hybrid as well as random test generation techniques using various fault models. Our experimental results demonstrate that our proposed technique is scalable, and enables directed test generation for large designs.**

I. Introduction

Functional verification is very important during the development of modern System-on-Chip (SoC) designs. As the complexity of embedded systems grows exponentially in recent years, it is becoming increasingly difficult to reach the desired coverage goal within the time budget using random or constrained-random tests. Directed tests can be used to alleviate this problem since the same coverage goal can be achieved with a significantly less number of directed tests. However, since most directed tests are written manually by experienced test engineers for important and corner case scenarios, it is hard for testing team to catch up with today's rapid development cycles. It is desired to have a fully automated way to generate directed tests.

Model checking is a promising technique to automatically generate directed tests [1], [2], [3]. Model checkers usually accept models presented in special verification languages. It is not easy to apply them on real implementations. For example, while the Register Transfer Level (RTL) model of real processors are commonly written in Verilog or VHDL, model checking tools like NuSMV only takes SMV models as input. Thus, real designs must be translated first, which itself can be an error-prone process. Since model checking is based on static analysis, the complexity of real world designs usually exceeds the capacity of model checking tools. The solving process may run out of memory before producing any useful results for real life designs. On the other hand, random or constrained-random test generation techniques are suitable for real designs, because they usually perform little reasoning on internal design logic. A large amount of random stimuli can be generated easily and simulated on real designs. However, random tests are inefficient to activate specific behaviors. It is therefore desirable to have a test generation approach that can handle real-life designs, but still able to activate any required system behavior with a small number of tests.

To bridge the gap between model checking based directed tests and random tests, we propose a novel test generation approach based on interleaved concrete and symbolic execution. Recent approaches [4] demonstrated that concolic testing [11] can be effectively utilized for hardware designs. However, existing approaches have major drawbacks e.g., they do not support dynamic array references. The applicability of the existing methods is limited to simple designs since dynamic array references are widely used in modern RTL designs to implement register files, buffers, caches and memory. Instead of performing static analysis of the entire design, we compose an instrumented version of the original design and execute the instrumented design on a Hardware Description Language (HDL) simulator. During the simulation, the instrumented code will produce a trace file, which records all the logical operations performed by the design. Next, the trace is analyzed using a constraint solver. In this way, our approach is able to analyze real hardware designs with dynamic array references and detect data dependency through array elements.

Our experimental results demonstrate that our approach is capable of generating directed test efficiently on a variety of hardware designs. To the best of our knowledge, our approach is the first attempt to create directed tests for HDL designs with dynamic array references by interleaving concrete and symbolic simulation. The rest of the paper is organized as follows. Section II describes related work on test generation techniques. Section III describes our test generation methodology for real HDL designs. Section IV presents our experimental results. Finally, Section V concludes the chapter.

II. Related Work

Simulation based validation is able to handle designs at different abstraction levels and therefore widely used in

This work was partially supported by National Science Foundation (NSF) grants CNS-0746261 and CCF-1218629.

practice. Genesys-Pro test generator [5] from IBM extended this direction with complex and sophisticated test templates. Wagner et al. [6] designed the MCjammer tool which can get higher state coverage than normal constrained random tests. Due to random nature, it is time consuming for a random test generator to activate all desired behaviors of the design.

On the other hand, model checking techniques are promising for automated generation of directed tests [7], [8]. Due to the state explosion problem, conventional symbolic model checking approaches are not suitable for large designs. SAT-based bounded model checking (BMC) is introduced by Biere et al. [9] as an alternative solution. Chen et al. [1] proposed directed test generation based on high level specification using SAT-based BMC. To accelerate the test generation process, conflict clauses learned during checking of one property are forwarded to speed up the SAT solving process of other related properties. These techniques rely on certain language for system modeling, and cannot take HDL design directly as input.

Directed test generation tools based on interleaved concrete and symbolic execution, such as DART [10] and CUTE [11], are promising in capturing important bugs in large software systems. STAR and HYBRO [4] are proposed to generate tests by combining static and dynamic analysis for hardware validation. Due to the effective utilization of the CFG, HYBRO [4] demonstrated remarkable improvement over previous path-based test generation technique. However, since the CFG and use-define chain is obtained using static analysis, this technique cannot be applied when dynamic array references [12] are involved. As a result, the applicability of [4] is limited to simple designs without dynamic array references, since dynamic array references are widely used in modern RTL designs to implement register files, buffers, caches and memory.

III. Directed Test Generation by Interleaving Concrete and Symbolic Execution

The basic idea of our work is to obtain the logic operations performed by the design on a single concrete execution path, and perform reasoning on top of it to obtain new test input. Figure 1 shows the key steps in our proposed approach. To explore different execution behaviors of the design, we first instrument the design with trace generation code. We also define the input variable set I, which present the input to the design under test (DUT). Next, we repeatedly simulate the instrumented design as follows:

1) Use I as input to the DUT.
2) Simulate DUT on a simulator. Collect all the operations performed by the design and activated path constraints from the trace output.
3) Invoke the constraint solver to check whether the desired behavior p is on current execution path. If this is the case, record the assignment of I as a test of p. Otherwise, negate one of the path constraints and use the constraint solver to obtain the assignment I', which forces the design to exercise a different execution path.

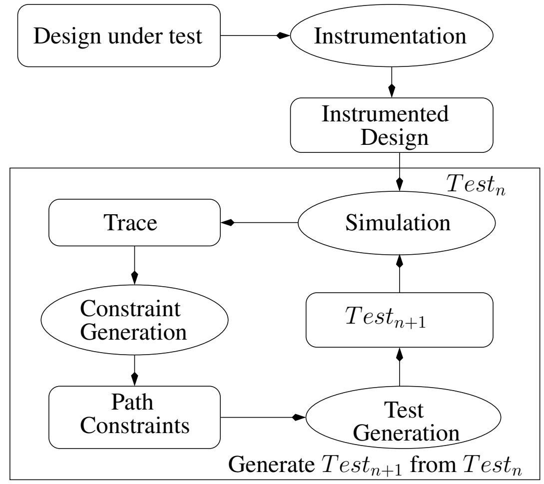

Fig. 1. The workflow of our approach

We first explain our test generation workflow using a simple example. Next, we describe the system model of our target design and several key steps in our workflow. Finally, we discuss some important optimization techniques to reduce the overall test generation time.

A. Illustrative Example

In this section, we use a simple example to show the basic workflow of our approach. The design is a simple counter module written in Verilog (Figure 2). The test input is the initial value on line 17. Our goal is to let the module execute the code on line 12 at clock cycle 2.

```
1   module counter(clk, reset);
2     parameter WIDTH = 8;
3     input  clk, reset;
4     reg [WIDTH-1 : 0]   out[8];
5     reg [2 : 0]   i;
6     wire   clk, reset;
7     always @(posedge clk)
8     begin
9       out[i+1] <= out[i] + 1;
10      i <= i+1;
11      if(out[i] == 40)
12        $display ("Activated");
13    end
14    always @reset
15      if (reset)
16      begin
17        out[0] = 0; // initial value
18        i = 0;
19      end
20  endmodule
```

Fig. 2. Counter.v

We first instrument the code and simulate the module for 3 cycles using a random initial value, e.g., out[0] = 0. The output trace is shown in Figure 3. The trace is produced by the instrumented code, which performs the same operations as the original code. In addition, the instrumented code also prints the performed operation during simulation as a trace file. We use (out[0],0), (out[0],1), (out[0],2) and (out[0],3) to represent each *out[0]* in different cycles. Notice that "IF (out[0],0) == 40 not taken" statement indicates that the if statement on line

11 is evaluated to be false. Clearly, line 12 is not executed when the initial value of out[0] is 0.

```
(out[0],0) = 0
(i,0) = 0
(out[1],1) = (out[0],0) + 1
(i,1) = (i,0) + 1
IF (out[0],0) == 40 not taken
(out[2],2) = (out[1],1) + 1
(i,2) = (i,1) + 1
IF (out[1],1) == 40 not taken
(out[3],3) = (out[2],2) + 1
(i,3) = (i,2) + 1
IF (out[2],2) == 40 not taken
```

Fig. 3. Sample Trace

Since our goal is to let line 12 to be executed at cycle 2, the variable *out[2]* must have value 40 at cycle 2. Similarly, (out[0],0), (out[1],1), (out[2],2) and (out[3],3) must satisfy constraints in Figure 4. Therefore, we can use constraint solvers like Yices [13] to solve these constraints, and produce the satisfiable assignments to all variables. In this case, the solver determines (out[0],0), (out[1],1), and (out[2],2) should be 38, 39, and 40, respectively. In other words, the initial value of (out[0],0) should be 38 in order to activate line 12 at cycle 2. This is the intended directed test.

```
(out[1],1) = (out[0],0) + 1
(out[0],0) != 40
(out[2],2) = (out[1],1) + 1
(out[1],1) != 40
(out[3],3) = (out[2],2) + 1
(out[2],2) = 40
```

Fig. 4. Sample Path Constraints

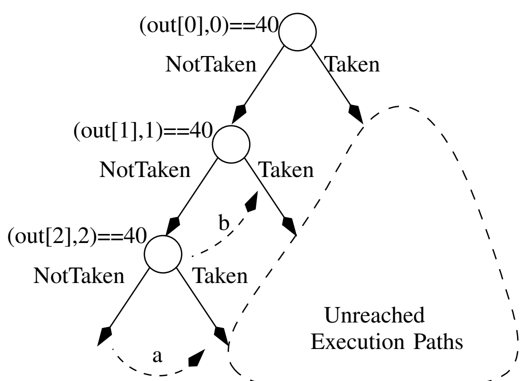

Fig. 5. Chronological Back Tracking

If we observe the path constraints (Figure 4) obtained from trace during concrete simulation (Figure 3), it is easy to see that we are essentially performing a chronological back tracking in the space of execution paths. By negating the topmost constraint[1] in the trace file ((out[2],2) != 40), we force the design to switch to a different execution path (transition "a" in Figure 5). Sometimes, it is also possible that our desired path constraints (Figure 4) are not satisfiable, i.e., the branch (out[2],2)==40 in Figure 5 can not be taken. In this case, we can negate the next topmost constraint ((out[1],1) != 40) and use the constraint solver to check whether the branch at node (out[1],1)==40 can be taken. The process is repeated, until the desired test is found, or all branches are activated.

It should be noticed that this example also contains dynamic array reference in line 9 and line 11. In different cycles, variable "out[i]" refers to different elements in the array "out". Therefore, it is difficult for techniques based on static analysis like [4] to detect such dependency. However, since we replace the array indices by their concrete values, such dynamic array references can be easily processed within our framework.

Although this test generation example is performed on a simple Verilog design, it illustrates the basic idea of our proposed approach. In the rest of the section, we are going to discuss how to automate every step during this process, and generate the entire directed test suite automatically.

B. System Model

Our approach takes Verilog HDL program as input. Our current implementation supports most common features of Verilog, such as always@(... sensitive list ...), continuous assignment, conditional branches (if, case), and different variable types (reg, wire). Although our implementation is based on Verilog, the same working principle can be applied to VHDL designs, since it also describes concurrent finite state systems.

Our current implementation supports common fault models such as path activation fault and stuck-at fault. These fault models describe possible faults that can occur during the execution of the system. The path activation fault model can be used to check whether there is any unreachable code in the design. The stuck-at fault can be used to check whether the given variable always has the same value. Based on the given fault model, our test generation technique will generate the test suite, which can activate all possible faults of the system under the fault model.

Without loss of generality, we discuss our approach in the context of single clock domain. We use tuple $(name, clk)$ to index each variable in every cycle. When multiple clock domains are used, the *clk* should be the cycle number in corresponding clock domain.

C. Instrumentation

The primary purpose of the instrumentation is to use the simulator to produce a trace file during concrete simulation of the RTL design. The resultant trace file is crucial to our test generation framework for two reasons. First, the trace file records all logic operations performed during the concrete simulation, which enables us to perform symbolic simulation and directed test generation. Besides, the trace file also provides information about different concrete execution paths. To ensure that each variable is unique, we need to flatten all module instances before instrumentation.

Table I shows the instrumentation rules. For ease of presentation, we use Verilog syntax for illustration. We use variable cc to denote the number of clock cycles from the beginning of the simulation. We use the "display" statement to print the syntactic objects into the trace file during the simulation of the instrumented code. For normal arithmetic operations,

[1]Due to use of stack in our implementation, the last path constraint is the topmost constraint.

TABLE I
VERILOG INSTRUMENTATION CODE

//Continuous assignment assign $v = e$;	always # clkwidth \$display($(v, cc) = e$); assign $v = e$;
//Blocking assignment $v = e$;	 \$display($(v, cc) = e$); $v = e$;

//Assignment within
//always@(pos/negedge ...)
$v <= e$;
	
	\$display($(v, cc+1) = e$);
	$v <= e$;

//Assignment within
//other always blocks
$v <= e$;
	
	\$display($(v, cc) = e$);
	$v <= e$;

//If
if(p)
s;
else
s';
	
	if(p)
	begin \$display(IF p taken); s; end
	else
	begin \$display(IF p not taken); s'; end

//Case case(e) x: s; y: s'; default: s'';	case(e) x: begin \$display(CASE $e = x$); s; end y: begin \$display(CASE $e = y$); s'; end default: begin \$display(CASE $e! = x, y$); s''; end

//Array index
//$b[e]$ is an array reference
// in a statement
...$b[e]$...;
	
	\$display(...b[$eval(e)$]...);
	
	...$b[e]$...;

//Beginning of a cycle	\$display(New cycle); $cc = cc + 1$;

the instrumented code just record the exact operation that is performed by the design. For example, for continuous assignment (first row in Table I), the original code is

```
assign x=y+z;
```

The instrumented code is

```
always
# cycle $display((x,cc)= (y,cc)+(z,cc));
assign x=y+z;
```

which have the same funtionality as the original code and print $(x, cc) = (y, cc) + (z, cc)$ in every cycle with corresponding cycle number (cc). In fact, the value of cc are populated automatically during concrete simulation.

For other assignment statements, the instrumented code also marks whether the assignment is made within always@(pos/negedge ...) block. In this way, the trace file records whether the left hand side variable receives the value of right hand side expression in the same clock cycle.

Our framework enables natural analysis of arrays. To reason with dynamic array references, we replace the index expression of each array elements into its concrete value, and treat each array element as an independent variable. During the concrete simulation, the index expression e is evaluated. The corresponding array element is refereed by concatenating the concrete results $eval(e)$ to the array name in the trace file.

D. Concrete Simulation

Once the design is flattened and instrumented, we interleave concrete and symbolic simulation of the design. In each iteration, we perform the concrete simulation of the instrumented design using a simulator with desired number of cycles. Since the instrumentation process does not affect the functionality of the design, the behavior of the instrumented design is identical to the original design. The instrumented design produces a trace file, which records every operation performed by the design in the correct order. This trace file will be used for the symbolic simulation of the concrete execution path.

E. Path Constraint Generation

In this step, we convert the trace file into a path constraint file. This step is required for two reasons. First, the continuous assignments are simulated using always blocks. As a result, the constraint corresponding to the continuous assignment may be printed after the trace is produced by the real always block in the same cycle. To simplify the solving process, we re-arrange the trace file so that all constraints produced by continuous assignments are placed before the constraints corresponding to normal always blocks.

The semantics of a register variable requires that if a variable is not updated, it should keep its value from the previous cycle. However, this property is not enforced by the constraints in the trace file. Thus, we have to examine that all assignments made during a cycle, and add additional constraints to ensure that all registers still maintain their values if they are not updated. The structure of a valid path constraint file is shown in Figure 6.

...	...
Cycle k	Continuous Assignments
Cycle k	Additional Constraints
Cycle k	Always blocks
Cycle $k+1$	Continuous Assignments
Cycle $k+1$	Additional Constraints
Cycle $k+1$	Always blocks
...	...

Fig. 6. Path constraint file structure

F. Test Generation

First, we discuss the test generation for path activation fault. Since the goal is to explore unreached execution paths, we can negate a path constraint and use the constraint solver to create a new input assignment, which will guide the design to a different path. Currently, we negate the top most path constraint. As a result, we are essentially performing a depth first search.

Algorithm 1 presents our test generation algorithm $test\_gen$ for path activation fault in detail. The algorithm takes the path constraint file $constr[0, ..., top]$ as input, where $constr[0]$ and $constr[top]$ are the first and last constraints in the file, respectively. Function $test\_gen$ examines all constraints produced by branch conditions in the reverse order. For every branch constraint, we first mark it as covered, then try to find the next uncovered branch constraint. For IF statement, we just need to check the negated version of the branch constraint. For CASE statement, we have to search for the next uncovered case. After that, the new branch constraint c is added to all previous path constraints $constr[0, ..., i-1]$ to form the constraints for the next test. If it is satisfiable, the assignment I' (returned from

Algorithm 1: Test Generation Algorithm

$test\_gen(constr[0, ..., top])$

for $i = top$ to 0 **do**
  if $constr[i]$ is a branch constraint **then**
    $c = find\_next(constr[i])$
    while $c \neq null$ **do**
      $I' = satisfy(constr[0, ..., i-1] \wedge c)$
      if $I' \neq null$ **then**
        return I'
      end if
      $c = find\_next(constr[i])$
    end while
  end if
end for
return $null$

$find\_next(branch)$

Add $branch$ into $covered$
if $branch$ is an IF statement **then**
  if $\neg branch \subsetneq covered$ **then**
    return $\neg branch$
  end if
end if
if $branch$ is a CASE statement **then**
  find and return next uncovered case, if any.
end if
return $null$

the constraint solver) will be returned as the next test input. Otherwise, we examine the next uncovered branch, until all branches are checked. In this way, it is guaranteed that I' will force the design to exercise a different execution path during the next round of simulation. Recall that the design is simulated for a fixed number of cycles. Our algorithm eventually terminates once all reachable branches within the given number of cycles are explored.

Each branch is uniquely identified by its line number, flattened instance name, and cycle number. To avoid the path explosion problem, a covered branch is marked and not explored again in the following test generation process.

For other fault models (including stuck-at, node, edge, sequence and interaction fault model), the desired behavior can be checked during the exploration of different execution paths. Once we obtain a new execution path, the constraint solver is employed to check whether the desired behavior is possible on the path.

G. Constraint Solving Optimization

In our current implementation, we employed Yices [13] as our constraint solver. Since the path constraint usually contains a very large number of constraints, it is very important to reduce time consumption in constraint solving. Currently, we use three optimization techniques.

Cone-of-influence (COI) reduction: In many designs, a large number of variables are used for data transfer and not involved in the control path.In other words, they are not in the cone-of-influence of any branch constraints in current execution path. It is therefore safe to remove the constraints involving these variables from the path constraint file without changing its satisfiability. This optimization is similar to the CFG unrolling and UD chain slicing technique proposed in [4]. It should be noticed that since the variable indices in arrays are replaced by their concrete values in the trace file, we are able to detect the data dependency through dynamic array reference.

Early unsatisfiable detection: Some variables, like reset signal, are used widely across the entire design as switch variables. As a result, they appear in the path constraint for several times in every clock cycle. It is enough to negate the first occurrence of a recurring path constraint, because the negation of its other occurrence must be unsatisfiable.

Unsatisfiable core detection: Some constraint solver is capable to return the unsatisfiable core of a unsatisfiable model. Clearly, if all constraints in the unsatisfiable core remains in the path constraint file, the model must be still unsatisfiable.This information can be utilized to reduce the number of expensive constraint solver calls by skipping the negation of some path constraints.

IV. Experiments

We developed a prototype of our directed test generation framework. Our test generation tool takes a Verilog design as input and iteratively produces new tests. We have modified Icarus Verilog [14] for instrumentation with approximately 500 lines of C++ code. We also implemented a test generation engine (approximately 2000 lines of C++ code) to perform concrete simulation on the HDL simulator, analyze the trace file, generate path constraints and invoke the SMT solver. Our framework is fully automated and there is no need for manual intervention at any stage.

In this section, we present the experimental results of our case studies. We compared our approach with HYBRO [4] and the random test technique. The experiments are performed using RTL models from ITC99 and a processor design. We used Icarus Verilog as parser and simulator. Yices [13] was employed for constraint solving. All experiments were performed on 3GHz AMD Opteron Processor with 10GB memory.

A. Designs without Dynamic Array References

In this section, we compare the performance of our approach with HYBRO [4]. To make fair comparison, we choose the same ITC99 RTL models as [4], with same number of unrolled cycles and the same SMT solver. We only compare the branch coverage in our experiments, because the assertions used for functional coverage in [4] is not available.

Table II presents the experimental results. The first two columns indicates the design name and the number of unrolled cycles. The next four columns show the branch coverage rate and the time consumption of HYBRO [4] and our approach,

TABLE II
COMPARISON WITH HYBRO [4]

Bench mark	Unroll Cycles	HYBRO[4]		Our approach	
		Bran_Cov	Time	Bran_Cov	Time
b01	10	94.44%	0.07s	96.30%	0.55s
b06	10	94.12%	0.10s	96.30%	0.46s
b10	30	96.77%	52.14s	96.67%	24.61s
b11	10	78.26%	0.28s	81.82%	0.67s
b11	50	91.30%	326.85s	94.44%	270.28s
b14	15	83.50%	301.69s	98.95%	257.59s

respectively. The branch coverage rate is calculated using the same convention in [4], where unreachable default branches in "case" statement are also included. The results suggest that our approach has comparable performance with HYBRO [4] on these benchmarks. Comparable performance is expected because the cone-of-influence reduction employed in our approach is essentially equivalent to the CFG unrolling and UD chain slicing optimization in HYBRO [4]. Note that there are 8 ITC99 benchmarks that have arrays. Since [4] is not applicable on dynamic array references, we did not compare with them.

B. Designs with Dynamic Array References

This experiment was performed on Zet processor, which is an open source implementation of the 16-bits x86 instruction set architecture. When synthesized using FPGA, Zet processor can boot MS-DOS 6.22 and run Microsoft Windows 3.0. The processor is implemented using 5K+ lines of Verilog code, 289 continues assignments, 53 always blocks, 324 register variables and 666 wire variables. Both main memory and register file are modeled as arrays and addressed with variables.

Our goal in this experiment is to achieve high branch coverage in source code level. This is important because there are a large number of conditional branches in the opcode decode stage. Besides, since x86 instruction set has variable length binary encoding, it is not easy to invoke all branches in the design. The primary input of the design is the lowest 4 bytes of the memory space (0x00000-0x00003). Before executing the test, the processor only executes a jump instruction (to 0x00000) after reset. The design is simulated for 10 cycles. We compared with random tests since HYBRO [2] cannot handle Zet processor that uses dynamic array references.

TABLE III
COMPARISON WITH RANDOM TESTING

Method	#Tests	Explored Branches	Branch Coverage	Time
Random	1000	197	89.95%	366.45s
Random	5000	204	93.15%	1981.73s
Random	10000	208	94.98%	3785.49s
Random	20000	212	96.80%	7386.92s
Random	40000	213	97.26%	14585.83s
Our approach	140	218	99.54%	1320.58s

Table III shows the experimental results. The first five rows depict the results by using 1000, 5000, 10000, 20000, and 400000 random tests, respectively. The performance of our approach is shown in the last row. The last column presents the total time consumptions for test generation and RTL simulation. It can be seen that due to the random nature, it is very time consuming to reach 100% branch coverage even using thousands of random tests. On the other hand, our directed test generation scheme effectively explored execution paths by avoiding covered branches. With less than 200 tests, our framework achieves higher coverage than 40000 random tests.

V. Conclusion

Directed tests generated using model checking are promising for functional verification, because they require significantly less number of tests to achieve the same coverage goal compared to random tests. Unfortunately, model checkers usually do not accept real hardware designs or support features such as arrays. Moreover, the real designs usually exceed the capacity of model checkers due to the complexity of static analysis. In this paper, we presented a novel test generation approach that addresses both of these problems using interleaved concrete and symbolic execution. The design is first simulated to generate an execution trace. The constraint solver is then applied to find the test inputs which can force the real design to exercise the desired behavior. Compared with existing approaches based on combined concrete and symbolic execution, our approach is capable of analyzing real processor designs with dynamic array references. The experimental results demonstrate that our proposed technique is scalable, and enables directed test generation for real designs.

References

[1] M. Chen, X. Qin, H. Koo and P. Mishra, *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*, Springer, August 2012.

[2] X. Qin and P. Mishra, "Efficient directed test generation for validation of multicore architectures," in *ISQED*, 2011, pp. 1–8

[3] X. Qin, M. Chen and P. Mishra,"Synchronized generation of directed tests using satisfiability solving," in *VLSI Design*, 2010, pp. 351–356

[4] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *DATE*, 2011, pp. 1–6.

[5] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 84 – 93, 2004.

[6] I. Wagner and V. Bertacco, "Mcjammer: adaptive verification for multicore designs," in *Proceedings of DATE*, 2008, pp. 670–675.

[7] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proceedings of the European Software Engineering Conference*, vol. 24, 1999, pp. 146–162.

[8] X. Qin and P. Mishra, Directed Test Generation for Validation of Multicore Architectures, *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 3, article 24, 2012.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999, pp. 193–207.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of PLDI*, 2005, pp. 213–223.

[11] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Proceedings of ICCAD*, 2006, pp. 419–423.

[12] D. Bernstein, D. Cohen, and D. Maydan, "Dynamic memory disambiguation for array references," in *Proceedings of International Symposium on Microarchitecture*, 1994, pp. 105 – 111.

[13] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proceedings of CAV*, 2006, pp. 81–94.

[14] S. Williams, *Icarus Verilog*, Icarus Verilog, 2012. [Online]. Available: http://iverilog.icarus.com/