Fast Multi-Level Test Generation at the RTL

Kelson Gent and Michael S. Hsiao
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
{kelsong, mhsiao}@vt.edu

***Abstract*—Functional, at-speed vectors continue to provide added value to the testing community as circuit complexity rises. Complex defects may escape traditional scan vectors and thus often require at-speed patterns. However, generation of functional/sequential vectors is an extremely challenging problem. Previous methods rely on formal models of the RTL or calls to gate level ATPG, both of which are computationally expensive, limiting the efficacy of gains made in RTL stimuli generation. In this work, we present an efficient engine for the generation of high quality functional tests at the RTL which are effective for both validation and at-speed defect detection. The proposed method utilizes a rule based, behavioral coverage metric to accurately assess the activation of circuit modules by the generated stimuli. Based on this metric, we are able to effectively generate functional test vectors at the RTL, without additional gate level information, that can achieve a high level of defect coverage and up to an order of magnitude speedup over existing techniques.**

I. Introduction

Due to the growth in circuit complexity, conventional test methods, such as scan ATPG, is no longer sufficient to test complex and transient defects that manifest in modern designs. These defects require high-quality at-speed tests to adequately cover[1] leading to functional tests being used more frequently for testing across the entire design process [2, 3]. However, high quality functional/sequential automatic test pattern generation (ATPG) is extremely computationally expensive, particularly for large circuits. To manage this complexity, RTL ATPG has become a significant area of interest to generate such vectors to cover these cases. Vectors that simultaneously provide high levels of coverage for RTL verification and at-speed coverage stand to yield significant savings in the time and monetary costs of test and verification.

Several techniques for generating sequential vectors have been proposed at the RTL. Early RTL methods, such as PRINCE [4] utilize line coverage as their primary coverage metric. A prior approach at the RTL [5] represents the RTL as assignment decision diagrams to utilize formal techniques for test generation. A hybrid model was proposed in [6] using a polynomial circuit model to leverage simulation alongside the use of formal models. However, these techniques frequently fail to reach deep, narrow states due to the significant overhead imposed by their use of bounded model checking. Significant advances in RTL state justification have been made utilizing simulation-based and hybrid models. BEACON [7], utilized an evolutionary ant colony optimization to generate functional verification vectors targeting branch coverage. PACOST [8] uses a formal model to generate an onion-ring guidance model for simulation. In [9] a data-mining based approach to learn about cross cycle transitions which are used to guide the test generation towards specified target states. However, all these methods utilize macro level code coverage metrics, such as branch coverage, which do not adequately represent lower level behavior within the design. To combat this, mixed level generation is used in [10–12], these methods generate vectors at the RTL and then refine the generation process using information from gate level simulation or test generation. However, the invocation of gate-level fault simulation significantly slows these methods and the full benefit of generation at the RTL is not realized.

To address the deficiencies in previous methods, we propose a swarm intelligence based vector generation engine utilizing a fine grained RTL coverage metric [15]. Originally proposed for vector grading, we adapt the metric to target the generation of functional tests for both design verification and at-speed defect test. The coverage metric is based on bitwise behavioral coverage of RTL operators in the HDL source. Using a set of rules based on the value of the operands, a set of coverage points is updated based on the value of the operands during simulation. In this work, operator coverage points serve as an abstraction for potential defect activation during the search. By using this metric as a heuristic basis for ATPG, we can assess areas in the RTL with low levels of behavioral activation. Using an Ant Colony Optimization(ACO) to target these areas during search, we can achieve significant improvements in defect coverage for functional test patterns generated at the RTL, while still providing quality vectors for design verification. Additionally, we do not incur the high cost of formal methods calls or gate-level fault simulation, minimizing the time required for test generation.

Our test generation algorithm is formulated as an Ant Colony Optimization (ACO) meta heuristic using the operator coverage metric as its base heuristic. Initially, the circuit is compiled to a fast, optimized cycle-accurate C++ model and instrumented via a set of monitors. These monitors provide complete coverage of all intermediate signal values within the single cycle model RTL. Then, during test generation, each ant generates a sequential test vector. The quality of the test vector is analyzed based on the fine-grain operator coverage metric. A fitness score for each ant is assessed by the level of coverage provided by the test vector. As the coverage of a code block increases, additional coverage within that block drops in value, allowing for the specific targeting of poorly exercized blocks within the description. The coverage of deep states in the state space provides additional propagation paths and access to defects that would otherwise be undetectable.

The contributions can be summarized as follows:

supported by NSF grant 1422054

- We present an ATPG engine with a novel guidance metric that utilizes a fine grain RTL coverage metric to generate quality functional test vectors at the RTL.
- The fine grain coverage metric allows for the targeting of uncovered blocks during test generation to achieve high level of behavioral activation in each module in the circuit.
- Our method provides significant performance improvements over previous techniques by operating entirely at the RTL. The gain is realized through the elimination of computationally expensive gate-level simulation and formal methods during generation.

The rest of the paper is organized as follows. Section II discusses relevant past work and fundamental theories, including the ACO and coverage metric. Section III covers search algorithm and the utilization of the RTL coverage metric. Section IV discusses the performance of the ATPG engine compared to previous works. Finally, Section V provides the concluding summary.

II. Background and Preliminaries

In this section, we outline the prior works used during test generation, as well as techniques used for analyzing RTL descriptions. We additionally provide a description of the coverage metric and the rules for generating the metric from the RTL.

A. Ant Colony Optimization

The ACO [13] is a biologically inspired algorithm that models graph search as a foraging simulation of an ant colony. Each individual search unit is modeled as an ant that communicates information through the use of pheromone trails. This communication acts as the mechanism for reinforcement learning of the swarm and can be used as a meta heuristic for NP-hard search problems. Specifically, in a graph G, edges denotep paths that ants can traverse. Starting from an initial location, a population of ants begins a random walk through G. At each transition they make their decision based on a set of parameters: $pheromones(\phi)$ and $visibility(\psi)$. Pheromones are left by each ant in the colony, based on how favorable the transition was between two vertices in the graph. As ants pass edges, they prefer paths with large amounts of pheromones, because these paths have been evaluated well by other ants within the colony. This produces a system of *reinforcement* among the ants. In order to avoid convergence to a local, non-optimal solution, *evaporation* is added to the system. This globally reduces the amount of pheromones on each edge at regular intervals to allow for promising new paths to better compete with existing paths.

B. Control Flow Graphs

The Control Flow Graphs (CFG), proposed by Allen [14], provides the basis for many compiler level optimizations and static analysis tools. The CFG is represented as a directed graph $G(V, e)$ with vertices representing basic blocks and edges representing the flow of execution between basic blocks. Each basic block is the maximal number of program statements such that it meets the following conditions:

1) Each block can only be entered via the first statement.
2) Each block may only contain one exit statement that leads to another basic block.
3) All statements must execute sequentially within a block.

Graph edges are created based on the execution targets of the final statement in the block to form the CFG. Based on this analysis, loop optimization and unreachable program segments can be determined and eliminated during compilation.

C. RTL Coverage Metric

The metric is based on the observation that within the RTL, each conditional statement represents a potentially unique logic path created during synthesis. Therefore, conditional statements create critical barriers for propagation. These points can block fault propagation to either state or output variables. An example of a blocking control structure created during synthesis is shown below in Figure 1. In this example, if x never equals $1'b1$; all faults from the 'and' gate will never be propagated across this gate. Due to these controlling structures, global metrics such as toggle, transition or signal value coverage are insufficient to accurately gauge defect coverage. To minimize this discrepancy, the coverage points for each statement in the RTL are generated based on behavioral rules for each operator. We provide an overview of the method [15] below.

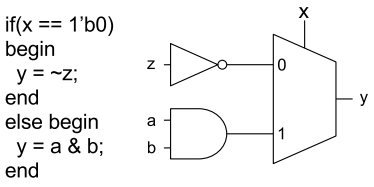

Fig. 1. Control Statement Synthesis

1) Operator Coverage Rules: The metric encapsulates the behavior of each operator as a set of coverage points. The coverage points are generated from a set of behavioral rules which include a scheme to partition the inputs to the operator and a set of critical observation values. The different operators are shown below in Table I.

TABLE I
Operator Types

Linear and Bitwise	$\pm, +, -, \&, \mid, \sim, \wedge, >>, <<, reduction$
Nonlinear	$\times, \div, \%$
Select Operators	$[\,], [:]$
Conditional Operators	?: , $<, \leq, >, \geq, ==, \neq$, &&, $\|\|$

Linear and bitwise operators have straightforward coverage point generation. The bitwise logical operators are direct representations of gate level behavior and synthesis must reflect that behavior. Therefore, they are modeled as a set of two input logic gates connecting the operands. Other linear operators are covered using behavioral coverage applied to the partitioned operands. For example, an adder is covered as a set of single bit additions. Binary shift operators are partitioned into individual bits and covered as simple bit value coverage. However, if the digits to be shifted are all one or zero, the shift operand is

ignored. The rules for partitioning and critical coverage values are shown below in Table II.

TABLE II
LINEAR OPERATOR OVERVIEW

Ops	Partition	Coverage Values	# Cov. Pts.
$+, -$	$\{X_i, Y_i\}$	$00, 01, 10, 11$	$4 \cdot min(W_x, W_y) + \mid W_x - W_y \mid$
$\&$	$\{X_i, Y_i\}$	$11, 01, 10$	$3 \cdot min(W_x, W_y) + \mid W_x - W_y \mid$
$\mid$	$\{X_i, Y_i\}$	$00, 01, 10$	$3 \cdot min(W_x, W_y) + \mid W_x - W_y \mid$
$\sim$	$\{X_i\}$	$0, 1$	$2 \cdot W$
$\wedge$	$\{X_i, Y_i\}$	$01, 10, 11 or 00$	$3 \cdot min(W_x, W_y) + \mid W_x - W_y \mid$
reduction	$\{X_i\}$	$0, 1$	$2 \cdot W + 2$
$>>, <<$	X_i, Y_i	$0, 1$ & $0, 1$	$2 \cdot W + log_2(W)$

W is the bit-width of the operand. X and Y are the operands of the operations.

To evaluate non-linear operators, coverage points are created using an expanded form of the operator. The expansion for multiplication is based on a technique for self-verifying multipliers [16] with a low probability of fault escape.

Access operators, with the exception of array access, are treated as masked signals. Therefore, a weak toggle coverage is used to generate vectors that are sensitive to accidental changes of the masking values. However, for array logic, the correctness must be ensured of the selection logic. These memories are categorized into two types, RAM and register files, denoted by their size. Arrays of size less than 8192 bytes are considered to be registers or lookup buffers. Table III outlines the properties of the non-linear and select operators.

TABLE III
NON-LINEAR AND ACCESS OPERATOR OVERVIEW

Ops	Partition	Values	# Cov. Pts.
$\times$	mod3	$00, 01, 10$	$3 \cdot (W_x/2 \cdot W_y/2)$
$[], [:]$	X_i	$0, 1$	W

III. METHODOLOGY

In this section, we describe the framework of the proposed test generation algorithm and the integration of the coverage metric described above into the swarm intelligence based search. Although the ACO meta-heuristic has been used in the past, the fitness functions and swarm dynamics in our method operate quite differently.

A. Additional Operator Coverage

In addition to the coverage metric described in Section II-C, we add new coverage points for conditional operators within branching statements. To generate these coverage points, we utilize techniques based on domain coverage [17]. The condition is treated as a boundary between two domains, true and false. Conventional domain testing aims to properly exercise the condition such that tests near the boundary that have a high likelihood of a domain change in the presence of a defect. We extend this concept to emphasize that domain changes due to defects do not only occur along the boundary but also at specific ranges based on the conditions.

For the two different types of logical operators, Boolean arithmetic and numerical comparison, different techniques are used for measuring the corresponding coverage. For Boolean operators, the bitwise counterpart for each single bit is used. To adequately target numerical operators, a fault on any given bit should cause a behavioral change. Therefore, two conditions must apply for a bit change to be considered exercised. First, the change must cause the result of the comparison operator to change value. Second, the change of operator evaluation must change the evaluation of the entire conditional expression. These two properties ensure that any defect will cause a behavioral change along the circuit control path. An example is shown below in Figure 2 for the conditional expression $x <= 5$ when the value of x is set to 0110. Two bitwise domain crossings are shown for $x = 0110$ flipping the second and third least significant bits in the representation. Note, that when the second least significant bit is flipped, x becomes 0100, or 4, changing the outcome of the conditional $x <= 5$. A similar analysis can be performed for the second most significant bit, where 0110 is changed to 0010. In this case, the result will also alter the outcome of the conditional $x <= 5$. However, for covering the least significant bit, a different value for x must be used, since flipping the least significant bit of 6 (i.e., 0110 to 0111) will not alter the outcome of the conditional.

In our new metric, when all bits of variable involved in a conditional is covered, we guarantee that a change to any bit position in that variable is captured by at least one test.

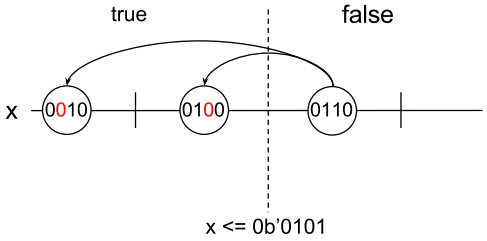

Fig. 2. Bitwise Domain Coverage

For the conditional assignment operator, coverage points are created for just the control statement, which is sufficient to ensure that each possible assignment has been exercised. A summary of these additional coverage points is given below in Table IV.

TABLE IV
CONDITIONAL OPERATOR OVERVIEW

Ops	Partition	Values	# Cov. Pts.
$\wedge, \vee$	X_0, Y_0	$0, 1$	4
$<, \leq, >, \geq, =, \neq$	X_i	$0, 1$	$2 \cdot W_x + \cdot W_y$

B. HDL Preprocessing

In order to process the circuit for test generation, we must first analyze and generate meta-information to calculate our coverage metric. We first translate the Verilog RTL to a cycle-accurate C++ simulation library using Verilator[18]. During the conversion, we extract and output the control flow data graph (CDFG) of the circuit under test. Based on the CDFG, we inject a set of monitoring functions into the RTL at each control flow and data dependency edge as SystemVerilog Direct Programmable Interface(DPI) calls. An example CDFG with injected monitors is shown below in Figure 3.

Following monitor injection, each statement is registered with a monitor. When called, the monitors capture the current state of the simulator and a set of unique identifiers for the

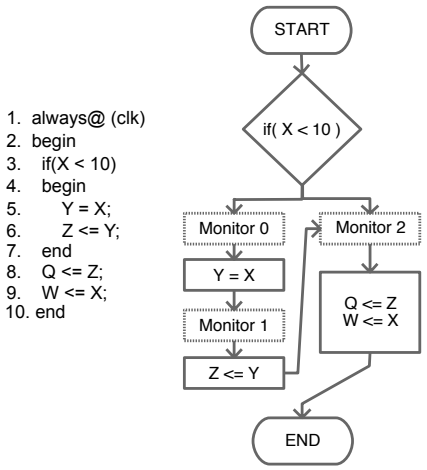

Fig. 3. CDFG Based Monitor Injection

registered assignments. Then, the DPI function updates the coverage database based on passed circuit state. This platform allows for direct access to simulator internals with the need for custom code for each circuit. It also provides a flexible method for introducing additional analysis code into the verilated simulator. The preprocessing flow is shown in Figure 4.

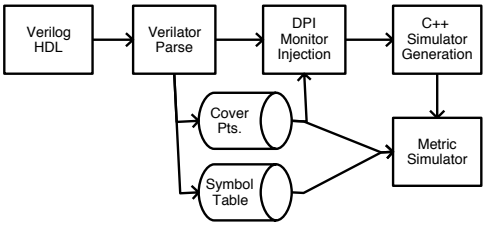

Fig. 4. Verilator Compilation with DPI Injection

C. Test Generation

Test vector generation is implemented using an ACO meta-heuristic algorithm. The colony is a pheromone database ϕ, and a set of K ants. Each ant in the swarm is represented by a test vector initialized with random stimuli, a set of local operator coverage points and a Verilated simulator instance. Additionally, the colony reads the single cycle CDFG used to inject monitors in phase to gain additional information about the circuit structure. Each ant the simulates its associated vector from the colony starting state S_0, typically circuit reset. Following generation, each vector is scored according to the coverage reached during simulation and ants lay pheromones based on the quality of the inputs generated. Then, we update the relative value of each coverage point according to the number of remaining uncovered points in the control flow block. Additionally, we increase the number of cycles in the ant generation to attempt to target deep code blocks that may need long sequences to activate. Then, a new round begins and the colony begins search based on the updated path weights. This process continues until the system uncovers no new behavior in N_r rounds. At this point, the algorithm assumes the system has reached steady state within N_c cycles from S_0 and terminates. The pseudo code for the ACO is shown below in Algorithm 1.

Algorithm 1 ATPG Engine

```
initialize pheromone map φ
analyze CDFG
for all rounds r = 1 to N_r do
    set the initial states to nest S_0
    ant_vector_generation()
    if all points covered || set_s = ∅ then
        RETURN
    else
        Update path weights
        Extend N_c
        S_0 = select(set_s)
        set_s = ∅ //clear the stack
    end if
end for
```

D. Ant Vector Generation

The ant vector generation process is shown in Algorithm 2. During vector generation, each ant within the colony attempts to generate vectors that mimic known good behaviors within the swarm-based on the pheromones deposited along the CDFG. For each cycle within the search, an ant starts by generating a random vector. The vector is simulated and the execution path taken within the CDFG is determined based on which monitors were activated. If the path activates high value code blocks, then the vector is kept and added to the ants generated vector. However, if only low value paths are executed, then the ant attempts to generate a new vector based on information learned by prior ants. This process continues until a high value path is found or the local generation times out. Following the vector generation, pheromones are updated based on coverage behavior observed during the cycle.

E. Pheromone Deposit along CDFG Path

The pheromone database represents the relative value of the observation of each path in the single cycle execution path seen by the swarm. During vector generation, the pheromone trail is used as a positive feedback mechanism to guide the swarm towards desirable behavior. Following the simulation of the test vector, the database is updated via reinforcement and evaporation as described in the following sections.

1) Reinforcement: Pheromone Reinforcement is done in two stages. First, if new coverage points or code blocks with CDFG edges to unexecuted code blocks are activated during simulation, pheromones are deposited based on the value of executing the path with regards to the operator coverage metrics. Then, additional pheromones are added if the execution leads to high value executions in the next cycle.

When paths are executed, they are given a key within the database and an initial amount of pheromones are deposited along the path. In future executions of the path this amount is adjusted based on the percentage of uncovered points for

Algorithm 2 Ant Search with Targeted Input Generation

```
S_branch = uncovered_blocks()
for all ants k = 1 to K do
    for all cycles c = 1 to N_c do
        generate_input()
        evalutate_path()
        while n_gen < TIMEOUT do
            regenerate_input()
            evaluate_path()
            if path pheromones > threshold then
                break
            end if
            ++n_gen
        end while
        deposit pheromones
    end for
end for
update_pheromone()
if new blocks covered then
    n = 0 // clear the rounds counter
else
    ++n
end if
```

statements along the path. The equation for this is given by:

$$\phi_{path}(t+1) = \phi_{path}(t) + ((N_c - N_h)/N_c) \cdot Q$$

where $\phi_{path}(t)$ is the amount of pheromones currently on the path, N_c is the number of coverage points associated with statements along the path and N_h is the number of covered points along the path. Basing the quantity of pheromones deposited on the amount of remaining coverage has two primary benefits. First code blocks devalue over time as their coverage increases, therefore, the algorithm will naturally trend towards attempting to execute paths with low levels of coverage. Second, paths with fewer statements will devalue faster than blocks with many statements, providing guidance on the importance of particular paths to defect coverage.

Second, additional pheromones are deposited on the path taken within the same always block in the prior cycle, if the pheromones on the current path exceed a threshold value. This additional deposit is given by the formula below:

$$\phi_{priorpath}(t+1) = \phi_{priorpath} + Q_{secondary}$$

This additional creates linkages between execution paths across cycles and aids the generation algorithm in navigating narrow execution paths by valuing paths whose statements already have a high level of statement coverage if they lead to desirable executions in the next cycle. This is necessary to the success of the algorithm as an RTL behavioral description may have paths with very few statements that are easily covered, yet, are critical to the advancement of the finite state machine towards hard to reach states.

2) Evaporation: For each path currently in the database, after all ants have been executed, we perform evaporation via the following equation:

$$\forall \phi_{path} \in database : \phi(t+1) = \phi(t) \cdot (1-\rho)$$

Where $0 < \rho < 1$ is the evaporation rate.

Evaporation allows for high value paths to devalue over time and not overly influence the test generation process. Additionally, evaporation prevents from following paths that saturate early from dominating the test generation process. Additionally, it allows paths to degrade that have not yielded any additional coverage information.

3) Pheromone Initialization: During the initial random simulation, pheromones are not deposited. However, we calculate operator coverage for the initial randomly generated vectors. This initialization creates a baseline value for each code block by trimming coverage points that are easily covered by random simulation. This initial value aids the ACO target harder to reach paths by reducing the amount of pheromone that would be deposited along the paths that are easy to test randomly.

IV. Experimental Results

The effectiveness of the proposed method is evaluated on a subset of the ITC99 benchmarks [19] and the OpenRISC 1200 processor [20]. We demonstrate the value of the method on couple of fault models, non-scan transition and non-scan stuck at, as well as provide the level of branch coverage achieved by the metric. The non-scan stuck-at fault coverage is compared to the mixed-level generation in [12] and a high level test generation tool, HTest [5]. The characteristics of the benchmarks are shown in Table V.

TABLE V
Benchmark Characteristics

Benchmark	Lines	PIs	POs	FFs	Logic Gates
b10	210	12	6	17	155
b11	131	8	6	30	353
b12	614	6	6	121	987
b13	361	11	10	53	289
b14	1030	33	54	247	3375
b15	750	36	70	447	6826
or1200	14695	164	207	2234	31144

A. Experimental Setup

Experiments were run on a single core of a Intel i7-3770k@3.5GHz with 16 GB of RAM on Ubuntu Linux 14.10. The ant colony is initialized with $K = 100$ ants and a maximum number of iterations is set at $R = 10$. N_c is initially set to 3000 vectors and the expansion multiplier is 1.5. Pheromones are initially set to 0 and the base deposit rate is $Q = 100$. The secondary deposit rate $Q_{secondary}$ is set at 15. The evaporation rate, ρ, is set to 0.1. Finally, the time out for generating inputs that reach the pheromone threshold is set to 15 attempts.

B. Generated Vector Quality

The non-scan stuck-at fault coverage for our method is reported in Table VI. For stuck-at defects, our method is able to achieve high levels of coverage comparable to the

TABLE VI
NON-SCAN STUCK-AT FAULT COVERAGE RESULTS

Benchmark	HTest			BEACON			[12]			Ours		
	FC(%)	Time (s)	Size	FC(%)	Time (s)	Size	FC(%)	Time (s)	Size	FC(%)	Time (s)	Size
b10	**94.4**	17.58	2847	85.5	11.4	3547	91.4	1.5	3160	91.0	**0.66**	4157
b11	89.4	5.85	972	77.5	11.9	1235	**91.8**	7.2	5680	91.3	**0.72**	3004
b12	N/A	N/A	N/A	77.7	111.4	37006	**90.3**	184.0	327578	89.7	**22.8**	182641
b14	N/A	N/A	N/A	81.9	204.6	4381	**86.7**	424.3	310000	83.9	**37.2**	204245
b15	N/A	N/A	N/A	77.0	255.9	12917	**91.18**	893.6	132794	90.6	**59.8**	112087
or1200	N/A	N/A	N/A	41.4	**300.40**	7946	N/A	N/A	N/A	**59.7**	363.3	32030
Note: Generation times for HTest are reported from older platforms												

mixed-level generation algorithm, even for known difficult sequential circuits such as b12. Our metric also provides value over BEACON, which only considers branch coverage as it's metric. Once branch coverage saturates, there is still a large amount of potential behavior left uncovered. The algorithm also outperforms Gate-level sequential tst generators which generally cannot reach even 50% coverage for b12. The loss of 0.6% coverage in b12 represents a total loss of less than 20 stuck-at defects during the test simulation. However, we are able to achieve a $9\times$ speedup in execution time as well as a much shorter test vector. Additionally, by performing our search entirely at the RTL, we are able to limit the overhead to generate vectors targeting the entire or1200 circuit which was infeasible in previous techniques.

Our method also provides high quality coverage for other metrics such as at-speed transition fault coverage and branch coverage. In Table VII, we show the transition fault coverage of vectors generated by our method. Though some of the margins, such as b12 are larger than that of the stuck at model, the transition coverage is similar to prior algorithms. This means that small changes in observable coverage did not yield significantly different transition coverage. Therefore, given generally smaller vector lengths and high execution speed, the RTL based generation yields competitive results compared to the mixed level generator. Additionally, or1200 still yields significant areas for improvement. Much of the OR1200 circuit used is difficult to test due to the SOC packaging. External interfaces such at the PWM and wishbone memory controllers add significant levels of indirection to be able to handle as a full package. In future work, we hope to address these levels of indirection in testing large SoC type packages.

TABLE VII
BRANCH AND TRANSITION FAULT COVERAGE

Benchmark	Branch Coverage		Transition Coverage	
	[12]	Ours	[12]	Ours
b10	100	100	77.15	78.4
b11	97.9	97.9	80.2	79.6
b12	99.1	99.1	65.1	62.5
b14	93.4	93.4	71.3	70.7
b15	91.3	91.3	64.7	65.4
or1200	N/A	94.2	N/A	36.4

V. CONCLUSIONS

In this paper, we presented an algorithm for sequential ATPG at the RTL. The test generation process is formulated as an ant colony optimization using operator coverage as the primary heuristic. Pheromones are placed along the single cycle CDFG on discovery of new operator behavior. This method generates high quality vectors for use in both validation and at-speed functional test with a high level of design and defect coverage. The algorithm also shows significant performance gains by leveraging the available information at the RTL without requiring information from a gate-level netlist.

REFERENCES

[1] E. J. McCluskey and C.-W. Tseng, "Stuck-fault tests vs. actual defects," in *Proceedings of the 2000 IEEE International Test Conference*, ITC '00, (Washington, DC, USA), pp. 336–, IEEE Computer Society, 2000.

[2] L. Lee, L.-C. Wang, P. Parvathala, and T. Mak, "On silicon-based speed path identification," in *VLSI Test Symposium, 2005. Proceedings. 23rd IEEE*, May 2005.

[3] R. McLaughlin, S. Venkataraman, and C. Lim, "Automated debug of speed path failures using functional tests," in *VLSI Test Symposium, 2009. VTS '09. 27th IEEE*, May 2009.

[4] F. Corno, G. Cumani, M. Reorda, and G. Squillero, "Effective techniques for high-level atpg," in *Proc. Asian Test Symp.*, pp. 225–230, 2001.

[5] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential atpg for functional rtl circuits," in *Proc. Int. Test Conf.*, vol. 1, pp. 290–298, Sept 2003.

[6] M. Mirzaei, M. Tabandeh, B. Alizadeh, and Z. Navabi, "A new approach for automatic test pattern generation in register transfer level circuits," *IEEE Design & Test*, vol. 30, pp. 49–59, Aug 2013.

[7] M. Li, K. Gent, and M. Hsiao, "Design validation of rtl circuits using evolutionary swarm intelligence," in *Proc. Int. Test Conf.*, 2012.

[8] Y. Zhou, T. Wang, T. Lv, H. Li, and X. Li, "Path constraint solving based test generation for hard-to-reach states," in *Proc. Asian Test Symp.*, pp. 239–244, Nov 2013.

[9] K. Gent and M. S. Hsiao, "Abstraction-based relation mining for functional test generation," in *VLSI Test Symposium (VTS), 2015 IEEE 33rd*, pp. 1–6, April 2015.

[10] M. Reni Krug, M. Soares Lubaszewski, and M. de Souza Moraes, "Improving atpg gate-level fault coverage by using test vectors generated from behavioral hdl descriptions," in *Proc. VLSI Test Symp.*, pp. 314–319, Oct 2006.

[11] S. Ravi and N. Jha, "Fast test generation for circuits with rtl and gate-level views," in *Proc. Int. Test Conf.*, pp. 1068–1077, 2001.

[12] K. Gent and M. Hsiao, "Dual-purpose mixed-level test generation using swarm intelligence," in *Test Symposium (ATS), 2014 IEEE 23rd Asian*, pp. 230–235, Nov 2014.

[13] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Tran. Systems, Man, and Cybernetics*, vol. 26, pp. 29–41, Feb 1996.

[14] F. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, pp. 1–19, July 1970.

[15] K. Gent and M. S. Hsiao, "A control path aware metric for grading functional test vectors," in *Latin American Test Symposium (LATS), 2016 IEEE 17th*, April 2016.

[16] M. Yilmaz, D. Hower, S. Ozev, and D. Sorin, "Self-checking and self-diagnosing 32-bit microprocessor multiplier," in *Test Conference, 2006. ITC '06. IEEE International*, pp. 1–10, Oct 2006.

[17] Q. Zhang and I. G. Harris, "A domain coverage metric for the validation of behavioral vhdl descriptions," in *Test Conference, 2000. Proceedings. International*, pp. 302–308, 2000.

[18] "Verilator." http://www.veripool.org/wiki/verilator.

[19] S. Davidson, "Itc99 benchmark circuits - preliminary results," in *Proc. Int. Symp. Circuits & Systems*, p. 1125, 1999.

[20] "OpenRISC web page." http://www.opencores.org.